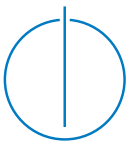
TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik

# From Smart to Secure Contracts: Automated Security Assessment and Improvement of Ethereum Smart Contracts

Christof Ferreira Torres



Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

| | | |
|---|---|---|
| Vorsitzender: | Prof. Dr. Sjouke Mauw | |
| Prüfende/-r der Dissertation: | Prof. Dr. Claudia Eckert | Prof. Dr. Radu State |
| | Prof. Dr. Jens Grossklags | Prof. Dr. Sjouke Mauw |
| | Prof. Dr. Shweta Shinde | Prof. Dr.-Ing. Hugo Jonker |

Die Dissertation wurde am 14.02.2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 27.04.2022 angenommen.

*To my parents.*

*“A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.”*

– Douglas Adams in *Mostly Harmless (The Hitchhiker’s Guide to the Galaxy)*

# Abstract

Modern blockchains, such as Ethereum, gained tremendously in popularity over the past few years. What partially enables this large increase are so-called smart contracts. These are programs that are deployed and executed across the blockchain. However, just like traditional programs, smart contracts are subject to programming mistakes. Although, unlike traditional programs their code is publicly available and immutable. Hence, as smart contracts become more popular and carry more value, they become a more interesting target for attackers. In the past few years, several smart contracts have been exploited, resulting in assets worth millions of dollars being stolen. In this dissertation, we explore the security of smart contracts from three different perspectives: *vulnerabilities*, *attacks*, and *defenses*, and demonstrate that, as so often, "smart" does not imply "secure".

In the first part of the dissertation, we study the automated detection of vulnerabilities in smart contracts, without requiring prior access to source code. We start by building a symbolic execution framework for detecting integer bugs that leverages taint analysis to reduce false positives. However, as symbolic execution is notorious to produce false positives, we explore hybrid fuzzing as an alternative. As a result, we propose a hybrid fuzzer for smart contracts that combines symbolic execution with fuzz testing and leverages data dependencies across state variables to efficiently generate transaction sequences. Our approach is capable of detecting more vulnerabilities with less false positives.

In the second part of the dissertation, we explore various ways to mount attacks against smart contracts. We start by proposing a framework to detect and quantify classical smart contract attacks (e.g., reentrancy, integer overflows, etc.) on past transactions by combining logic-driven and graph-driven analysis. Afterwards, we study the effectiveness of a new type of fraud known as *honeypots*, by scanning the entire blockchain for different types of honeypots using symbolic execution. Next, we present a methodology to measure the prevalence of so-called *frontrunning* attacks, which follow from the rise of decentralized finance and the sharp increase of users trading on decentralized exchanges. Our results show that attackers are making a fortune by manipulating the order of transactions.

In the third and final part of the dissertation, we discuss several defense mechanisms for smart contracts. We first propose a solution that developers can use to automatically patch vulnerable smart contract bytecode using context-sensitive patches that dynamically adapt to the bytecode that is being patched. However, this does not solve the issue of already deployed smart contracts. To that end, we present a second solution that enables security experts to write attack patterns that are triggered whenever malicious control and data flows are detected. Once a transaction is detected to be malicious, all state changes are rolled back and the attack is thereby prevented. These attack patterns are written using a domain-specific language and are managed via a smart contract. The latter enables decentralization, guarantees the distribution of security updates, and warrants transparency.

# Zusammenfassung

Neuartige Blockchains, wie Ethereum, haben in den letzten Jahren enorm an Popularität gewonnen. Ermöglicht wurde dies unter anderem durch sogenannte Smart Contracts. Smart Contracts sind Programme, die auf der Blockchain ausgeführt werden. Ebenso wie herkömmliche Programme sind auch Smart Contracts anfällig für Programmierfehler. Im Gegensatz zu herkömmlichen Programmen ist der Code jedoch öffentlich verfügbar und unveränderbar. Aufgrund der zunehmenden Bedeutung und Popularität werden Smart Contracts auch zu einem interessanteren Ziel für Angreifer. In den letzten Jahren wurden bereits mehrere Smart Contracts ausgebeutet, wodurch bereits ein Vermögen im Wert von mehreren Millionen Dollar gestohlen wurde. In dieser Dissertation analysieren wir die Sicherheit von Smart Contracts mit dem Fokus auf drei verschiedene Aspekte: *Schwachstellen*, *Angriffe* und *Schutzmechanismen*; und zeigen, dass “smart”, wie so oft, nicht automatisch “sicher” bedeutet.

Im ersten Teil der Dissertation analysieren wir die automatisierte Erkennung von Schwachstellen in Smart Contracts, ohne dass ein Zugriff auf den Quellcode erforderlich ist. Zunächst erstellen wir ein Framework für symbolische Ausführung zur Erkennung von Integer-Bugs, welches Datenflussanalyse benutzt, um die Anzahl von falsch positiven Ergebnissen zu verringern. Da symbolische Ausführung jedoch dafür bekannt ist, falsch positive Ergebnisse hervorzubringen, untersuchen wir hybrides Fuzzing als Alternative. Wir stellen einen hybriden Fuzzer für Smart Contracts vor, welcher symbolische Ausführung mit dynamischer Ausführung vereint und Datenabhängigkeiten über Zustandsvariablen hinweg nutzt, um Transaktionssequenzen effizient zu generieren. Durch diese Vorgehensweise können mehr Schwachstellen mit geringerer Fehlerquote erkannt werden.

Im zweiten Teil untersuchen wir verschiedene Möglichkeiten, Angriffe auf Smart Contracts durchzuführen. Wir stellen ein Framework zur Erkennung und Quantifizierung klassischer Smart Contract-Angriffe (z.B. Reentrancy, Integer Overflows, etc.) auf vergangene Transaktionen vor, bei dem Logik und Graphen gesteuerte Analyse kombiniert werden. Anschließend analysieren wir die Effektivität einer neuen Betrugsart, welche als *Honeypots* bekannt ist, indem wir die gesamte Blockchain mittels symbolischer Ausführung nach verschiedenen Arten von Honeypots durchsuchen. Als nächstes präsentieren wir eine Methode zur Messung der Prävalenz sogenannter *Frontrunning*-Angriffe, welche aus dem Aufstieg des dezentralisierten Finanzwesens und dem enormen Anstieg von Nutzern, die an dezentralen Börsen handeln, folgen. Unsere Ergebnisse zeigen, dass Angreifer mit der Manipulation der Transaktionsreihenfolge ein Vermögen verdienen.

Im dritten und letzten Teil untersuchen wir mehrere Abwehrmechanismen für Smart Contracts. Zuerst stellen wir eine Lösung vor, mit der Entwickler anfälligen Smart Contract Bytecode mit Hilfe kontext-sensitiver Patches automatisch patchen können, welche sich dynamisch an den zu patchenden Bytecode anpassen. Dies löst jedoch nicht das Problem

bereits aufgesetzter Smart Contracts. Aus diesem Grund präsentieren wir eine zweite Lösung, welche es Sicherheitsexperten ermöglicht, Angriffsmuster zu beschreiben, die ausgelöst werden, wenn bösartige Kontroll- und Datenflüsse ausgeführt werden. Sobald eine Transaktion als bösartig erkannt wird, werden alle Zustandsänderungen zurückgesetzt und der Angriff wird dadurch verhindert. Diese Angriffsmuster sind in einer domänenspezifischen Sprache geschrieben und werden über einen Smart Contract verwaltet. Letzteres ermöglicht die Dezentralisierung und gewährleistet die Verteilung von Sicherheitsupdates, sowie Transparenz.

# Acknowledgments

I was extremely lucky to have met and enjoyed the support of many people during those past four years. First and foremost, I would like to thank Radu State, not only for his supervision and guidance, but also for his neverending optimism and sense of humor.

I am deeply grateful to Jean Hilger and Christophe Medinger, for giving me the opportunity to be part of Spuerkeess and to learn a plenitude about the financial sector. Thanks for always being so kind to me and giving me the support and freedom that I needed.

I would like to thank Hugo Jonker and Sjouke Mauw, for their everlasting commitment to my academic development and for always believing in my academic potential. Without you, I would not have chosen this path.

My deep gratitude also goes to Claudia Eckert, for co-supervising me and for giving me the possibility to pursue a cotutelle at the Technical University of Munich and hosting me at Fraunhofer AISEC.

I am also very grateful to the other members of the defence committee, Jens Grossklags and Shweta Shinde. Thank you for the valuable time and effort that you two have spent in reviewing my thesis.

This dissertation would not have been possible without the support of my colleagues from SEDAN: Mathis, Robert, Beltran, Ramiro, Antonio, Andres, Sean, Mary, Nino, Katya, Manxing, Fernando, Flaviene, Sasha, Eric, Jorge, Wazen, Farouk, Lucian, and Vito, just to name a few. You all gave me the feeling of being part of a big family. I want to especially thank Mathis, Robert, Beltran, Ramiro, and Antonio. I cherish the priceless time we had during the winter school in Interlaken, Mathis. I will never forget the funny night we had in Seoul at the food market with the lady that served us those spicy stir-fried rice cakes, Robert. I look back to the hilarious debates we had during lunchtime at the canteen, Beltran. I will always remember the delightful discussions we had during our coffee and cookie breaks, Ramiro. I think back to the nights we went out for food and drinks, Antonio. I hope that someday we will be able to all meet again in a bar and grab a beer (or two).

Next, I would like to thank my colleagues from Fraunhofer AISEC: Mathias, Konrad, Florian, and Christian. Thank you for the “Feierabend” beers and the lunch walks. I wish to also thank Julian Schütte, for his guidance and support.

Thank you Valérie, for shielding me from the university’s bureaucracy. Due to you, I never had to worry about administrative tasks. If you ever need me again to paint some walls, just drop me a message, I am more than happy to help. I still hope that someday we will be able to do BBQ at your place.

I would also like to thank Jessica Giro for not giving up on my hopeless case and helping me turning my cotutelle into a reality.

Finally, I thank Spuerkeess and the Luxembourg National Research Fund (FNR) for their financial support.

Last, but definitely not least, I would like to thank my family, especially my mom and dad, for their endless support, encouragement, care, and advice to follow my dreams. However, the person who I probably need to thank the most is you, Anica. You had to listen to all my stories, all my complaints, and all my ups and downs during those past four years. Thank you for being always there for me, for loving me, and being so patient with me.

*To each and every one of you, thank you for the wonderful time! It was epic!*

Christof FERREIRA TORRES

Munich, January 2022

# Contents

# List of Figures

# List of Tables

# 1 Introduction

It is September 15, 2008, when Lehman Brothers, a well established financial institution with a history of over 150 years and more than 25,000 employees worldwide, declares bankruptcy [29]. The bankruptcy filing involves more than 600 billion USD and is considered the largest bankruptcy filing in U.S. history. The liquidation of Lehman Brothers is just one of many that would follow during the so-called Great Recession. Set in motion by the U.S. housing bubble and the global financial crisis in 2007, the Great Recession marks a time in which a large number of people lost their jobs, homes, and in some cases even their lives [28]. As a result, many people also began to lose their trust in governments and especially in financial institutions. The world was going through a dark chapter.

On October 31, 2008, a user named *Satoshi Nakamoto*[1], posted a whitepaper to a cryptography mailing list, that described a fully decentralized electronic cash system that does not depend on trusted third parties such as governments or financial institutions [13]. This was the moment where *Bitcoin* was born, and its underlying technology: *blockchain*. What initially started as a cryptocurrency that was merely used by nerds and geeks to buy pizzas online [159], has grown to a cryptocurrency with a daily trading volume of more than 20 billion USD and a market capitalization worth almost 1 trillion USD [207]. Bitcoin was not the first cryptocurrency to be proposed [2, 7]. However, it was the first cryptocurrency that solved the double-spending problem of digital currencies in an exceptionally elegant way. While being accessible to everyone, it did not rely on a central authority, nor did it require participants to trust each other or to be online at all times.

Blockchain is often regarded as one of the most disruptive technologies since the invention of the Internet itself. In recent years, companies across the globe have poured significant value into blockchain research, examining how it can make their existing business more efficient and secure. A blockchain is essentially a verifiable, append-only list of records in which all transactions are recorded in so-called *blocks*. Every block, except the very first block, is linked to its previous block via a cryptographic hash, thereby forming a chain of blocks or a so-called “blockchain”. This list of records is maintained by a distributed peer-to-peer network of untrusted nodes, which follow a consensus protocol that dictates the appending of new blocks.

[1] Satoshi Nakamoto’s true identity remains a mystery until today.

A diverse range of blockchain implementations have emerged since the release of Bitcoin. All of these implementations pursue a common goal, namely decentralizing the control over a particular asset. They achieve this by substituting trusted central entities with a large network of untrusted entities who strive to reach consensus on a history of transactions. Trust is obtained via the assumption that the majority of these entities act faithfully and respect the blockchain protocol, since going against the protocol becomes too costly and therefore irrational. The asset that Bitcoin aims to decentralize, is its own cryptocurrency, and the trusted centralized entities it attempts to replace, are traditional banks.

However, things changed when in 2013, Vitalik Buterin contended that Bitcoin and its underlying technology could be used for other purposes besides decentralized payments [24]. This was the moment where *Ethereum* was born. Ethereum goes a step further than Bitcoin by decentralizing the computer as a whole instead of just decentralizing banks. Ethereum is nowadays ranked as the second largest cryptocurrency in the world in terms of market capitalization, right behind Bitcoin [207]. It has been recently valued to be worth over 500 billion USD, thereby surpassing the value of Visa and JPMorgan [221]. Ethereum is different from Bitcoin in many ways. Its most important novelty is its capability to execute so-called *smart contracts*. Smart contracts are essentially programs that are deployed and executed across the Ethereum blockchain via the so-called Ethereum Virtual Machine (EVM) [30]. The EVM is a purely stack-based virtual machine that supports a large set of instructions, which enable the execution of Turing-complete programs. By introducing the concept of smart contracts to the masses, Ethereum revolutionized the way digital assets are traded. Smart contracts are usually developed using a high-level programming language. Despite a large variety of available programming languages (e.g., Vyper [223], LLL [75], Bamboo [93], Obsidian [65], etc.), Solidity [133] remains the most prominent language for developing smart contracts in Ethereum. Independently of the chosen programming language, the high-level source code must be always translated into a low-level bytecode representation, before it can be deployed and interpreted by the EVM. In contrast to traditional programs, smart contracts cannot be updated and may carry assets that can easily be worth millions. Thus, programming mistakes that were never intended by the developer become now both irreversible and devastating.

At the beginning of May 2016, almost a year after Ethereum's release, the Ethereum community announced the inception of the first so-called Decentralized Autonomous Organization (also known as "TheDAO"). It was intended to function in a similar way to a venture capital fund, where entities would operate through a smart contract instead of centralized governing authorities. By removing these centralized authorities, costs would be reduced and more control and access would be provided to the investors. The idea of TheDAO was to allow anyone to pitch their project to the community and potentially obtain funding directly from TheDAO's smart contract. Anyone owning DAO tokens could then vote on projects and obtain rewards if the projects turned profitable. TheDAO had a creation period during which

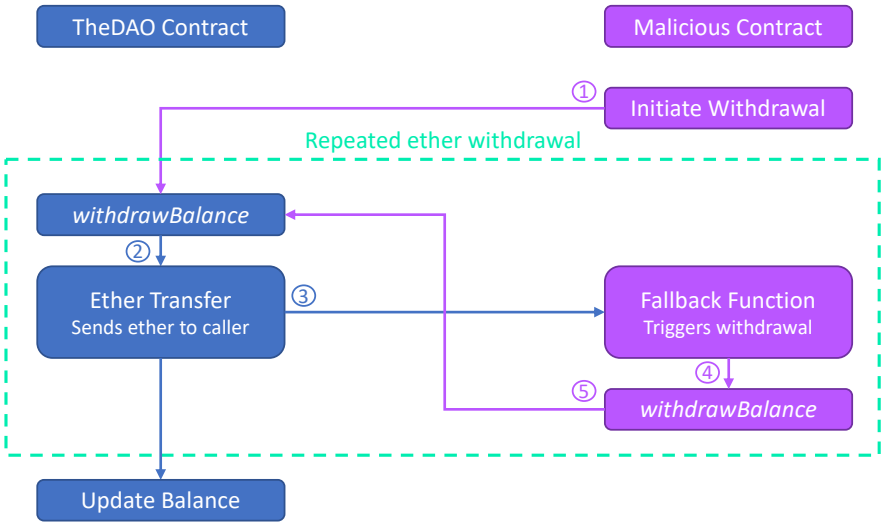



**Figure 1.1:** A visual example of the DAO reentrancy attack.

Ethereum users could send ether (Ethereum's own cryptocurrency) to a specific address in exchange for DAO tokens. The creation period surpassed everyone's expectations and became the biggest crowdfund ever by gathering 12.7M ether (worth around 150M USD at the time). With the funding in place, things were looking promising that the project would take off and become a huge success. However, on June 17, 2016, a user commented on Reddit that something odd would be going on with TheDAO's smart contract [49]:

> [...] *I think TheDAO is getting drained right now* [...]

It happened to be that an attacker found a loophole in the code of the smart contract, which allowed anyone to drain all funds from TheDAO. In the first few hours of the attack, 3.5M ether were stolen, the equivalent of 50M USD at the time. Interestingly, a week before the attack, on June 10, 2016, Christian Reitwiessner, one of the lead developers of the Solidity programming language, published a blog post about common smart contract security pitfalls [55]. In his blog post, the developer described exactly the same recursive call bug that was exploited during the DAO hack. A recursive call bug (also known as reentrancy) occurs when external contracts are allowed to call back (i.e., reenter) the calling contract before the initial execution is completed. Figure 1.1 provides an illustrative example of the reentrancy attack that was mounted against TheDAO. The attacker used a contract to call TheDAO's `withdrawBalance` function, which would then transfer ether to the calling address. In this case the calling address was a contract, which had been previously deployed by the attacker. An ether transfer to a contract always triggers the so-called *fallback* function. The attacker exploited this feature in order to call again (i.e., reenter) the `withdrawBalance` function of TheDAO's smart contract, before the balance was updated. The fact that the state of the smart contract is solely updated after the transfer and not before, allowed the attacker to

repeatedly withdraw ether from TheDAO. Ironically, on June 12, 2016 (5 days before the attack), the developers of TheDAO have been notified about the potential existence of such a recursive call bug within their smart contract, but the developers simply reassured the community that their smart contract would *not* be vulnerable [59]:

> [...] *No DAO funds at risk following the Ethereum smart contract 'recursive call' bug discovery* [...]

In the end, the developers were wrong, yet unbelievably lucky. The attacker decided to stop draining the smart contract, even though he or she could have continued to do so. This allowed a group of white hats to withdraw the remaining funds into a safe account. Moreover, any ether withdrawn from TheDAO's smart contract, would be first placed into a child DAO smart contract, which was subject to a 28 day holding period. The initial reaction of the community was to advocate a soft fork to stop the ether leaving the child DAO beyond those 28 days [37]. A soft fork does not result in a rollback (i.e., no past transactions or blocks are reversed), whereas a hard fork results in a rollback (i.e., past transactions or blocks are reversed). Unfortunately, despite having been integrated into most clients and having received major support from the community, the soft fork was canceled before it came into play as it would have opened up a denial-of-service vulnerability [48]. The last chance was a hard fork. The hard fork would move all the funds of the created child DAOs into a withdraw contract that would be available only to the original owners. The original owners would then be able to exchange their DAO tokens for ether. Despite being contested by many Ethereum users, the hard fork was successfully deployed on July 20, 2016, with only a slight majority voting in favor for the hard fork. A significant amount of users on the other hand, argued that the hard fork violated the basic principles of Ethereum and smart contracts: *"CODE IS LAW"*, meaning that code must be considered final once it is deployed on the blockchain and should not be changed. This dispute led to the Ethereum blockchain splitting into two separate chains. One that adopts the hard fork and which continued to be called "Ethereum", and another one called "Ethereum Classic", which in contrast to the other one, did not adopt the hard fork.

However, the DAO hack did not remain the only attack on Ethereum smart contracts. Since then, a number of other attacks have followed. Another prominent example are the two Parity wallet hacks. On July 19, 2017, an attacker found a vulnerability in the source code of Parity's multi-signature wallet smart contract. The attacker was able to steal over 150K ether (worth around 30M USD at the time) from a number of deployed contracts. To save deployment costs, Parity's wallets were using the same library contract. The idea behind a library contract, is to deploy redundant code only once on the blockchain. Other contracts can then call the library contract to execute the redundant code and thereby reduce the size of their own code. Thus, Parity's wallet contract would act as a proxy where the fallback function would forward unknown function calls via a delegatecall to the library

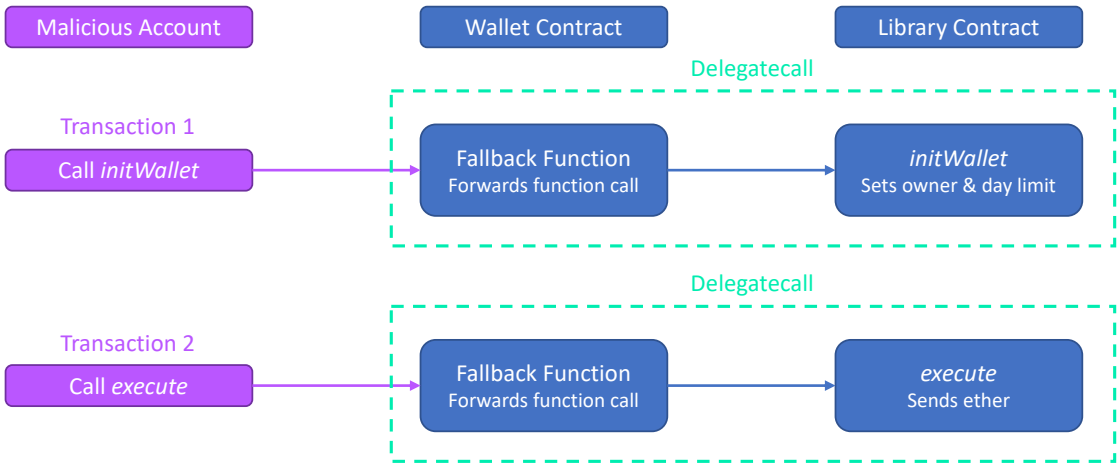



**Figure 1.2:** A visual example of the first Parity wallet attack.

contract. A delegatecall runs the code of the called contract under the context of the calling contract. This means that the called contract has access to the balance and storage of the calling contract. Unfortunately, Parity's library contract also contained functions that enabled everyone to set up the wallets. These functions should have been protected such that they could only be used under one specific circumstance, namely during the deployment of a new wallet contract. However, the functions were entirely unguarded, which allowed the attacker to reset the ownership and execute arbitrary functions. Figure 1.2 provides a visual example of the attack. The attacker had to send two transactions to the affected contracts. The first transaction would call the `initWallet` function on the wallet contract. The fallback function of the wallet contract would then forward the function call to the library contract and execute the `initWallet` function in the context of the wallet contract. The execution of `initWallet` would result in setting the attacker as exclusive owner of the wallet contract. The second transaction would call the `execute` function on the wallet contract. Similarly, the fallback function of the wallet contract would forward the function call to the library contract, however, this time execute the `execute` function in the context of the wallet contract. This would result in emptying the balance of the wallet contract and sending all the ether to the attacker.

A month later, Parity claimed to have fixed the bug and that it hired a security company to perform a security audit before deploying the new version of the library on the blockchain. However, the library contract was attacked a second time on November 6, 2017, just a few months after the first attack. This time, the attacker did not steal any funds, but publicly admitted of having accidentally killed the library contract (see Figure 1.3) and thereby destroyed the access to ether that is worth over 150M USD. Similar to the first attack, the second attack was composed of two transactions. Yet, in comparison, the attacker called in the second attack the library contract directly instead of going through a wallet contract and its delegatecall. Thus, the first transaction of the second attack directly called `initWallet` on the state of the library contract, instead of the wallet contract, and set the attacker as the


**Figure 1.3:** User admits on GitHub to have “accidentally” killed wallets belonging to Parity.

owner of the library. In the second transaction of the second attack, the attacker called the `kill` function of the library contract, which resulted in the library contract being destroyed (i.e., the code being removed and any subsequent calls to the library contract resulting in an error). Calling the `execute` function in the second transaction, as it was the case in the first attack, would make no sense since the execution context was the one of the library and the library did not contain any ether to be withdrawn. This second attack was possible because both, developers and auditors, always considered the contract to be used as a library and never thought that someone would directly call the library. Unfortunately, many Parity wallets had the address of the library contract hard-coded, with no option of updating it. As a result, the ether contained within these wallets is now lost forever, because the code of the destroyed library contract was the only way to move the ether out of the wallets.

## 1.1 Research Questions

The aforementioned incidents question the security of smart contracts. On the one hand, the capabilities offered by modern blockchains, such as Ethereum, fuel the development of new disruptive use-cases. On the other hand, the freedom of being able to write Turing-complete smart contracts entails new dangers and sparks a certain skepticism around the security guarantees offered by blockchains. Blockchains are often highly praised as secure and "unhackable". However, attacks such as the DAO hack [57] or the Parity wallet hacks [84, 80] demonstrated how vulnerable blockchain applications can be if the underlying smart contracts contain bugs. Due to these attacks, a lot of effort has been put in finding reentrancy and access control bugs in smart contracts, but only very little effort have been made to study whether smart contracts are vulnerable to traditional software bugs. For instance, integer overflows or so-called wraparounds are a popular type of bugs that are frequently found in traditional programs. In 2019, integer overflows have been ranked 8th in the top 25 most dangerous software errors [149]. Similar to traditional programs, integer operations are very common in smart contracts, for instance, to keep track of account balances.

Moreover, the way how Solidity and the EVM handle integer types may lead to unexpected border cases, thereby introducing vulnerabilities in smart contracts. This raises the question whether developers are aware of these peculiarities and if they add appropriate checks to their code. Bugs such as integer overflows, division by zero, or integer type conversion errors, belong to the family of so-called integer bugs. The accurate detection of integer bugs requires to know the accurate size (e.g., 32-bit or 64-bit) and sign (e.g., signed or unsigned) of an integer variable. This information may easily be retrieved from source code, depending on the programming language in which the program has been developed. However, a prominent issue when analyzing publicly available smart contracts for bugs, is the lack of source code. Ethereum for instance, only stores the bytecode and not the source code of a smart contract on its blockchain. This makes it hard to analyze smart contracts for integer bugs.

Most bug detection tools for smart contracts rely on symbolic execution (e.g., [51, 120, 116]). Symbolic execution has the advantage of being able to reason about all execution paths of a program. However, it also has several disadvantages. Symbolic execution is a static analysis technique, meaning that it does not execute code using concrete values but rather executes code in an abstract way, thereby replacing program inputs with symbolic values. This introduces approximation issues that often result in false positives. Moreover, symbolic execution is often combined with constraint solving in order to reason if an execution path is satisfiable or vulnerable under a given set of constraints. However, the ability of a constraint solver in solving a given set of constraints depends on the complexity of the constraints. In the context of symbolic execution, the complexity of the given constraints depends on the length of the execution paths and complexity of the branch conditions of a program. Symbolic execution is therefore prone to have difficulties in analyzing large complex programs. This is known as the “path explosion” problem. With the rise of complex use-cases, such as decentralized games and decentralized finance, smart contracts are becoming more and more complex and symbolic execution is reaching its limitations. In traditional software security, fuzzing or fuzz testing is a popular alternative to symbolic execution. Fuzzing is a dynamic analysis technique that executes a program using concrete values and checks if the execution path resulted in a bug being triggered. Fuzzing has the advantage of scaling well to larger programs and reporting no false positives (assuming that the bug detection is implemented correctly). However, the main challenge in fuzzing consists in generating meaningful inputs that explore all the different execution paths of a program. A popular way to generate meaningful inputs is to analyze the source code (i.e., white-box fuzzing). Unfortunately, as mentioned earlier, users mostly only have access to the public bytecode of a smart contract and therefore we must design tools that are capable of detecting vulnerabilities by only analyzing the bytecode of a smart contract. This leads to our first research question.

**Research Question 1 (RQ1)**

How can we detect vulnerabilities in smart contracts without having access to source code?

Over the past few years, industry as well as academia made a significant effort in releasing a number of vulnerability detection tools for smart contracts (e.g., [76, 102, 120, 87, 136, 112, 90, 155, 181]). Developers can use these tools to find and fix vulnerabilities in their smart contracts before deploying them on the blockchain. A large portion of these tools are publicly available and free to use (e.g., [76, 102, 87, 136]). Moreover, several smart contract auditing companies have been founded over the past few years (e.g., [217, 215, 206]) in order to provide professional services to developers by conducting manual security audits of smart contracts. It seems that checking smart contracts for vulnerabilities prior deployment is becoming a common practice. This raises the question whether these years of efforts have yielded visibly fewer attacks in the wild. If the tools proposed herein and the security audits performed by companies are effective, one could argue that attacks should have declined over time. Further, vulnerabilities such as reentrancy and faulty access control are often ranked as the most dangerous vulnerabilities in smart contracts [105]. However, this is most likely due to the significant monetary value that is often associated with these attacks. But what if other types of vulnerabilities are exploited more often, but just involve smaller amounts and therefore remain rather occluded?

Moreover, the security of smart contracts is often associated with the detection and exploitation of vulnerabilities contained within the code of a smart contract. However, smart contracts also heavily depend on a number of internal as well as external components. For example, smart contracts depend on the correct execution of the EVM or the process of correctly ordering transactions. These parts can be considered as internal components since they represent integral parts of the Ethereum blockchain. As external components, we may consider the Solidity compiler or blockchain explorers such as Etherscan. These are part of the Ethereum ecosystem but do not represent integral parts of the Ethereum blockchain. Independently of being internal or external, these components are as well just programs, meaning that they are inherently prone to programming mistakes. Thus, what happens if one or several of these components contain programming mistakes? Do these programming mistakes allow attackers to mount new types of attacks against smart contracts? This leads to our second research question.

**Research Question 2 (RQ2)**

What types of attacks exist leveraging smart contracts and how can we measure them?

Despite a large number of freely available vulnerability detection tools for smart contracts, developers are yet left alone with the burden of having to fix vulnerabilities manually. Existing tools often highlight the line at which a vulnerability resides, but developers are still required to know how to fix the vulnerability. Without appropriate knowledge or experience, developers might not be able to correctly fix the vulnerability, or even worse, their attempt may result in introducing new vulnerabilities. This became evident when the Parity wallet was hacked a second time after being manually patched following a security audit. Therefore, automatic patching offers a powerful promise to strengthen smart contract defenses, while at the same time minimizing the risk of introducing new vulnerabilities. However, current approaches either do not scale well to the growing complexity of smart contracts or are limited in the types of vulnerabilities that they can patch.

While automated vulnerability patching helps developers in fixing vulnerabilities before deployment, it does not solve the issue of protecting smart contracts that have already been deployed and contain vulnerabilities. Current solutions suggest that developers include, prior deployment, an option to either destroy their smart contract via a selfdestruct or upgrade their smart contract via a proxy contract. But what if a developer forgets to add such an option during development? Moreover, if a smart contract contains a bug and is under attack, then the developer is required to detect the attack and to manually react by quickly sending a transaction that temporarily disables the smart contract. However, the moment the developer realizes that their contract is under attack, it might be already too late and all the funds may have been drained out of the smart contract. This leads to our third and final research question.

**Research Question 3 (RQ3)**

How can we protect smart contracts against attacks before and after deployment?

## 1.2 Contributions

The main contributions of this dissertation are summarized as follows:

(1) We develop OSIRIS – a tool based on symbolic execution and constraint solving to study the prevalence of integer bugs in smart contracts. To reduce the reporting of benign integer bugs (e.g., introduced by the compiler for optimization purposes), OSIRIS leverages taint analysis to track only the flow of integer bugs that may be triggered by an attacker. We find that integer bugs are a prevalent issue in smart contracts. Our analysis on 1.2M contracts revealed that 42,108 contracts suffer from at least one integer bug (e.g., integer overflow, division by zero, truncation error, etc.). In comparison to existing works, OSIRIS achieves a considerably lower false positive rate. Moreover,

OSIRIS discovers major programming flaws in two contracts after analyzing the top 495 Ethereum token smart contracts. We also identify causes for integer bugs and propose possible modifications to the EVM and the Solidity compiler, to remove the burden from developers to protect their smart contracts against integer bugs.

(2) We present CONFUZZIUS – a hybrid fuzzer for smart contracts that does not require access to source code. CONFUZZIUS combines evolutionary fuzzing with constraint solving. We use evolutionary fuzzing to exercise shallow parts of a smart contract and constraint solving to generate complex inputs that satisfy conditions that allow the evolutionary fuzzing algorithm from exploring deeper parts of the contract. Moreover, CONFUZZIUS infers dynamically data dependencies across state variables to generate sequences of transactions that trigger complex smart contract states. We evaluate the effectiveness of CONFUZZIUS by comparing it with state-of-the-art symbolic execution tools and fuzzers. Our evaluation on a curated dataset of 128 contracts and a dataset of 21,147 real-world contracts shows that our hybrid approach detects up to 23% more bugs than state-of-the-art and that it outperforms existing tools in terms of code coverage by up to 69%. We also demonstrate that data dependency analysis can boost the detection of bugs up to 18%.

(3) We propose HORUS – an extensible framework for carrying out longitudinal studies on smart contract attacks. The framework identifies smart contract attacks by translating the execution of individual transactions into logical expressions which can then be queried using queries written in Datalog. In addition, the framework is capable to quantify the amount of stolen assets independently of their type (i.e., ether or tokens). Moreover, by loading transactions into a graph database, we can trace the flow of stolen assets across accounts and perform behavioral studies on attackers. We provide a longitudinal study on the security of Ethereum smart contracts by analyzing transactions of 4.5 years, ranging from August 2015 to May 2020. We find 8,095 attacks in the wild, targeting a total of 1,888 vulnerable contracts. Our analysis shows that on the one hand the number of attacks targeting integer overflows seem to have decreased over the years, but on the other hand attacks targeting unhandled exceptions and reentrancy seem to remain present despite a wealth of smart contract security tools. We also demonstrate the practicality of HORUS in identifying malicious transaction and tracing of stolen assets by applying it to the Uniswap and Lendf.me hacks that occurred in April 2020.

(4) We present a taxonomy of honeypot techniques and use this taxonomy to build HONEYBADGER – a tool that uses symbolic execution together with well-defined heuristics to automatically detect smart contract honeypots. Using HONEYBADGER, we conduct the first systematic analysis of honeypots by analyzing over 2 million smart contracts

deployed on the Ethereum blockchain, and confirm the prevalence of at least 690 deployed honeypots. An analysis on the transactions performed by a subset of the discovered honeypots reveals that 240 users already became victims of honeypots and that honeypot creators already made an accumulated profit of over 90,000 USD.

(5) We propose an efficient methodology to detect frontrunning attacks such as displacement, insertion, and suppression on Ethereum's past transaction history. We perform an extensive large-scale study on more than 11M historical blocks and identify a total of 199,725 attacks mounted by 526 bots and 1,580 attacker accounts. The measured attacks yield an accumulated profit of 18.41M USD for the attackers, providing evidence that frontrunning is both lucrative and a prevalent issue. Moreover, we are able to group the identified attacker accounts and bots into 137 unique attacker clusters. Finally, we also discuss frontrunning implications and find that miners are profiting from frontrunning practices by making a passive income of 300K USD only due to the high transaction fees that frontrunners have to pay to miners in order to be able to mount their attacks.

(6) We introduce a novel context-aware patching approach that combines template-based patching with semantic-based patching to create tailored patches for smart contracts. We build ELYSIUM – a tool that implements our approach to automatically patch 7 different types of vulnerabilities in smart contracts at the bytecode level. We compare ELYSIUM to existing works using 3 different datasets and by replaying more than 500K transactions. We not only demonstrate that ELYSIUM is capable of patching at least 30% more bugs than existing solutions, but that it is also more efficient in terms of gas consumption by using up to 1.9 times less gas.

(7) We propose ÆGIS together with a novel domain-specific language, which enables the definition of attack patterns. Attack patterns describe malicious control and data flows that occur during execution of malicious transactions. ÆGIS detects and reverts malicious transactions in real-time using these attack patterns, thereby preventing attacks on deployed smart contracts. Moreover, we introduce a new way to quickly propagate security updates without relying on client-side update mechanisms, by making use of a smart contract to store and vote upon new attack patterns. Storing patterns in a smart contract ensures integrity, decentralizes security updates and provides full transparency on the proposed patterns. We illustrate the effectiveness by providing patterns to prevent the two most prominent hacks in Ethereum, the DAO and Parity wallet hacks. We also provide a detailed comparison to current state-of-the-art runtime detection tools. Through a large-scale analysis on 4.5 million blocks, we demonstrate that ÆGIS achieves a better precision than current state-of-the-art tools.

## 1.3 Overview

This dissertation consists of three parts. Each part addresses one of the aforementioned three research questions. The first part is divided in two chapters and focuses on automated vulnerability detection of smart contract bytecode by employing static and dynamic analysis techniques. This part is based on two publications [102, 210], which are joint works with Julian Schütte, Antonio Ken Iannillo, Arthur Gervais, and Radu State. The second part is divided in three chapters and studies various types of attacks and frauds that can be mounted against smart contracts or using smart contracts. This part is based on four publications [158, 226, 211, 209], which are joint works with Antonio Ken Iannillo, Arthur Gervais, Mathis Steichen, Liyi Zhou, Kaihua Qin, Duc V Le, Ramiro Camino, and Radu State. The third and final part is divided in two chapters and proposes defenses for smart contracts, before and after deployment. This part is based on three publications [157, 180, 222], which are joint works with Hugo Jonker, Mathis Baden, Robert Norvill, Beltran Fiz Pontiveros, Sjouke Mauw, and Radu State.

**Part I: Detecting Vulnerabilities in Smart Contracts**

**Chapter 3** focuses on the detection of integer bugs in Ethereum smart contracts. This chapter addresses parts of RQ1 by leveraging symbolic execution in conjunction with constraint solving to analyze the bytecode of deployed smart contracts and to study the prevalence of integer bugs. This chapter is based on the paper:

- Christof Ferreira Torres, Julian Schütte, and Radu State. “Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts”. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pages 664–676, San Juan, PR, USA, December 3–7, 2018 [102].

**Chapter 4** studies the use of hybrid fuzzing for detecting bugs in Ethereum smart contracts. This chapter addresses parts of RQ1 by proposing a hybrid fuzzer for smart contract bytecode that solves the three challenges of smart contract testing: input generation, stateful exploration, and environmental dependencies. This chapter is based on the paper:

- Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. “ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts.” In *Proceedings of the 6th IEEE European Symposium on Security and Privacy (Euro S&P)*, pages 103–119, Virtual Event, October 7–22, 2021 [210].

**Part II: Studying Attacks on Smart Contracts**

**Chapter 5** analyzes the distribution of smart contract attacks on Ethereum over time. This chapter addresses parts of RQ2 by proposing a framework that first identifies attacks by leveraging logic-driven analysis to detect if the execution of a transaction exploits a given

vulnerability in a smart contract and afterwards identifies the flow of stolen assets by leveraging transaction graph based analysis. This chapter is based on the paper:

- Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. "The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts". In *Proceedings of the 25th International Conference on Financial Cryptography and Data Security (FC)*, pages 33–52, Virtual Event, March 1-–5, 2021 [211].

**Chapter 6** investigates the prevalence of smart contract honeypots on the Ethereum blockchain. This chapter addresses parts of RQ2 by employing symbolic execution to detect smart contract honeypots in the wild and by performing a longitudinal analysis on the effectiveness, liveness, behavior, diversity, and popularity of honeypots. This chapter is based on the paper:

- Christof Ferreira Torres, Mathis Steichen, and Radu State. "The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts". In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, pages 1591–1607, Santa Clara, CA, USA, August 14–16, 2019 [158].

**Chapter 7** measures the rising adoption of frontrunning practices on the Ethereum blockchain. This chapter addresses parts of RQ2 by presenting an approach that is efficient enough to analyze the information contained within historical blocks and measure the phenomenon of frontrunning as well as quantify the spread of the individual types of frontrunning: displacement, insertion, and suppression. This chapter is based on the papers:

- Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. "High-Frequency Trading on Decentralized On-Chain Exchanges". In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, pages 428–445, Virtual Event, May 23–27, 2020 [226].

- Christof Ferreira Torres, Ramiro Camino, and Radu State. "Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain". In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, pages 1343–1359, Virtual Event, August 11–13, 2021 [209].

**Part II: Proposing Defenses for Smart Contracts**

**Chapter 8** presents a framework to automatically patch vulnerable smart contracts before deployment. This chapter addresses parts of RQ3 by introducing a context-aware patching approach that combines template-based patching with semantic-based patching to create patches that are tailored to the implementation of each smart contract. This chapter is based on the paper:

- Christof Ferreira Torres, Hugo Jonker, and Radu State. “Elysium: Automagically Healing Vulnerable Smart Contracts Using Context-Aware Patching”. [222].

**Chapter 9** proposes a solution to defend against attacks on smart contracts after deployment. This chapter addresses parts of RQ3 by introducing a system that detects and reverts malicious control and data flows at runtime via attack patterns that are stored on the blockchain and which are written using a novel domain-specific language. This chapter is based on the papers:

- Christof Ferreira Torres, Mathis Baden, Robert Norvill, and Hugo Jonker. “ÆGIS: Smart Shielding of Smart Contracts (Poster)”. In *Proceeding of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2589–2591, London, UK, November 11–15, 2019 [157].
- Christof Ferreira Torres, Mathis Baden, Robert Norvill, Beltran Fiz Pontiveros, Hugo Jonker, and Sjouke Mauw. “ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks”. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (Asia CCS)*, pages 584–597, Virtual Event, October 5–9, 2020 [180].

**Chapter 10** finally concludes this dissertation by summarizing results and discussing limitations as well as future research directions.

# 2 | Background

In this chapter, we introduce the reader to the background that is necessary to understand the work conducted within this dissertation. We briefly highlight the technicalities of Ethereum and smart contracts including common vulnerabilities on smart contracts.

## 2.1 Ethereum

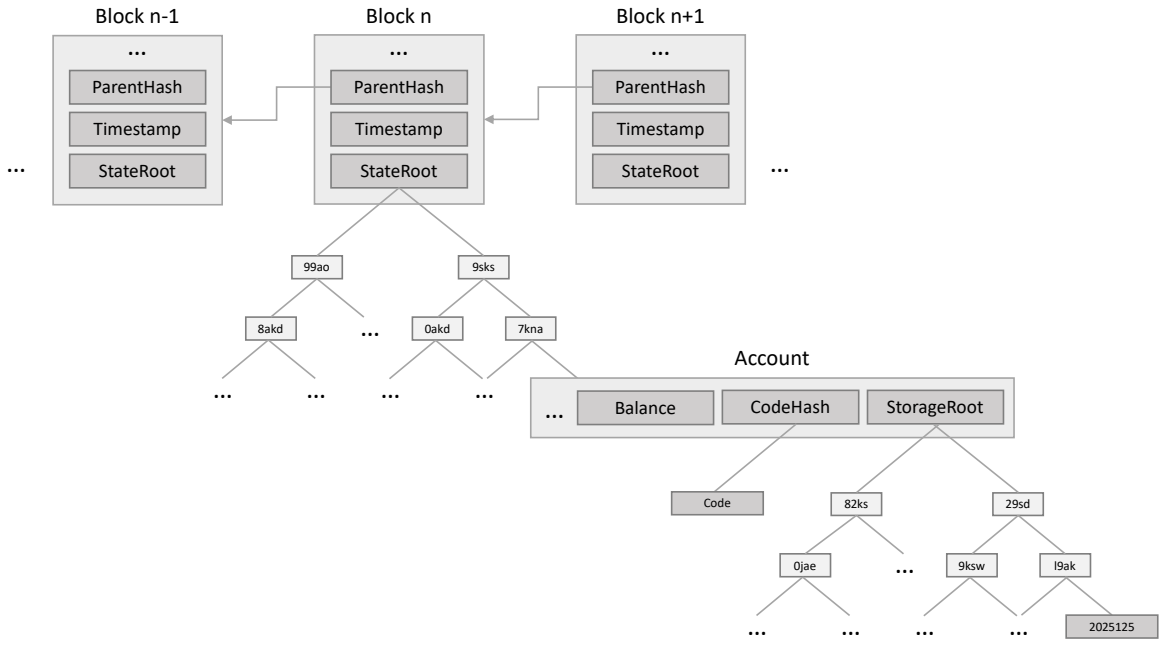



**Figure 2.1:** Ethereum blockchain structure.

Ethereum is a blockchain technology that was introduced in 2014 [30], with the first block being mined in July 2015. A blockchain is essentially a peer-to-peer network of computers that update and share a copy of the same database without necessarily knowing or trusting one another. The database acts as a ledger that keeps record of every single transaction that has been performed within this network. The word “block” refers to the fact that transactions are grouped into batches, which are called “blocks”. A transaction has to be included in such

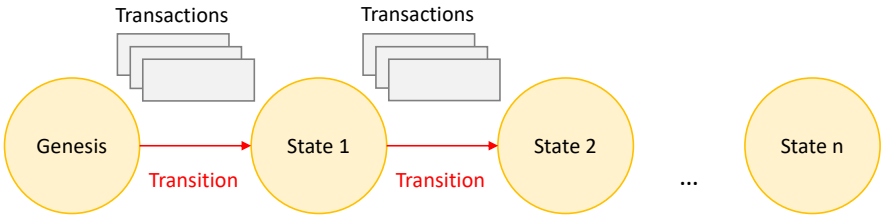



**Figure 2.2:** Ethereum can be seen as a transaction-based state machine.

a block in order to be considered valid. The word “chain” refers to the fact that each block is cryptographically linked to its previous block via a cryptographic hash, whereby the first block (i.e., genesis block) begin an exception as it has no previous block and therefore acts as a root of trust. In other words, blocks are chained together and form a chain of blocks, which is known as a “blockchain” (see Figure 2.1).

The Ethereum blockchain can be seen as a transaction-based state machine, that reads a series of inputs and, based on those inputs, transitions to a new state (see Figure 2.2). Ethereum starts with a blank state called the “genesis state”. This genesis state transitions into a new state after executing a batch of transactions. This new state represents the current state of Ethereum, at any point in time. In other words, blocks represent states and transactions represent state transitions. The data inside a block cannot be altered without changing all subsequent blocks. Changing all subsequent blocks would require the consensus of the majority of the network.

Every computer in the network must agree upon each new block and the chain as a whole. These computers are known as "nodes". Nodes acts as an entry point to the blockchain and ensure everyone interacting with the blockchain has the same view on the data. To accomplish this distributed agreement, blockchains make use of a consensus protocol. Ethereum currently uses Proof-of-Work (PoW) as its consensus protocol. In order to append a new block to the blockchain, users have to generate a hash of the new block, which starts with a given number of zeros. Finding such a hash requires a lot of computing power. It acts as a “proof” that a node has done “work” by spending its computational resources. This process is known as mining and nodes which decide to participate in this process are known as miners. Mining is essentially playing a lottery where miners follow a brute-force-based trial and error approach and the first to successfully mine a block is rewarded in the form of some cryptocurrency. New blocks are then broadcast to the nodes in the network, checked and verified, thus updating the state of the blockchain for everyone.

As nodes are allowed to propose new blocks at the same time, it can be the case that two or more blocks are proposed simultaneously with a valid hash while referencing the same parent block. This is called a “fork”. Forks pose a serious issue to blockchains as they result in multiple concurrent states (or chains) and make it hard to agree on which state is the correct one. For instance, if the chains were to diverge, a user might own 10 coins on

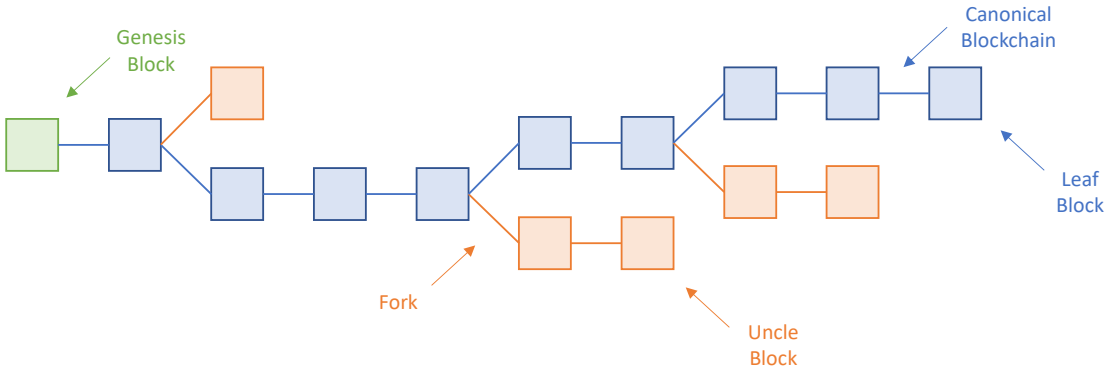



**Figure 2.3:** An illustration of Ethereum's GHOST protocol.

one chain, 20 on another, and 40 on another. To prevent multiple chains and help determine which fork is the most valid one, Ethereum uses a technique known as the Greedy Heaviest Observed Subtree (GHOST) protocol (see Figure 2.3). The GHOST protocol states that one must select the chain that contains the most computation. One way to determine that chain, is to leverage the block number of the most recent block (i.e., leaf block), which amounts to the total number of blocks in the current chain (not including the genesis block). The higher the block number, the longer the chain and the greater the mining effort that must have gone into arriving at the leaf. This allows nodes to agree on the canonical version of the current state. Blocks that are not included in the canonical chain are often referred to as orphans or uncles. In contrast to other blockchains, such as Bitcoin, Ethereum also adds uncle blocks to the calculation to figure out the longest and heaviest chain of blocks. This allows for the inclusion of more transactions and attributes a reward to the creators of uncle blocks as well as miners for declaring concurrent blocks as uncles and thereby keeping forked chains short.

### 2.1.1 Ether

Ether is the native cryptocurrency of Ethereum. A cryptocurrency is a digital currency that is secured by means of cryptographic primitives. The purpose of ether is not only to allow users to exchange value between one another, but also to provide an economic incentive for users to provide computational resources to the Ethereum network. Any participant who sends a transactions must offer some amount of ether to the Ethereum network as a remuneration. This remuneration will be awarded to whoever gets to mine the block that includes the transaction, as a result of doing the work of verifying, executing, and broadcasting the transaction to the rest of the network. The amount of ether offered must correspond to the time and effort spent in executing the transaction. These costs prevent malicious participants from intentionally congesting the network by requesting the execution of infinite computation or other resource-intensive actions, as these participants must pay for the computation. Ethereum

**Table 2.1:** List of cryptocurrency denominations in Ethereum.

| Denomination | Value in ether | Value in wei |
|---|---|---|
| wei | $10^{-18}$ | $1$ |
| Kwei (babbage) | $10^{-15}$ | $1,000$ |
| Mwei (lovelace) | $10^{-12}$ | $1,000,000$ |
| Gwei (shannon) | $10^{-9}$ | $1,000,000,000$ |
| microether (szabo) | $10^{-6}$ | $1,000,000,000,000$ |
| milliether (finney) | $10^{-3}$ | $1,000,000,000,000,000$ |
| ether | $1$ | $1,000,000,000,000,000,000$ |

provides a metric system of denominations to describe different units of ether (see Table 2.1). Each denomination has its own unique name (some bear the family name of figures that played an important role in cryptoeconomics and computer science). Wei and Gwei are the most popular denominations. Wei is the smallest possible denomination of ether also known as the base unit, and as a result, many technical implementations base all their calculations in wei. Gwei (short for gigawei) is often used to describe costs related to "gas".

### 2.1.2 Accounts

The global state of Ethereum is composed of many objects called "accounts". They are able to interact with one another through so-called "messages". Each account has a 20-byte address and a state associated with it. An address in Ethereum is a 160-bit identifier (a string of 42 hexadecimal characters) that is used to uniquely identify any account on the blockchain. There exist two different types of accounts (see Figure 2.4):

- **Externally Owned Accounts** (EOAs), which are controlled by private keys and have no code associated with them.
- **Contract Accounts** (CAs), which are controlled by their contract code and have code associated with them (i.e., smart contracts).

Both account types have the ability to receive, hold and send ether. EOAs can send messages to other EOAs and CAs by creating a transaction and signing it using their private key. The code that is associated with a CA, is executed whenever it receives a message from an EOA or a CA. The code allows a CA to perform various actions (e.g., write to storage, perform computations, etc.), which a EOA is not capable of. However, unlike EOAs, CAs cannot initiate new transactions on their own. Instead, CAs can only trigger messages in response to other messages that they have received from either an EOA or another CA. Thus, any actions that occurs on the Ethereum blockchain, are always set in motion by transactions that are triggered by EOAs. The account state consists of four fields, which are always present regardless of the type of account:

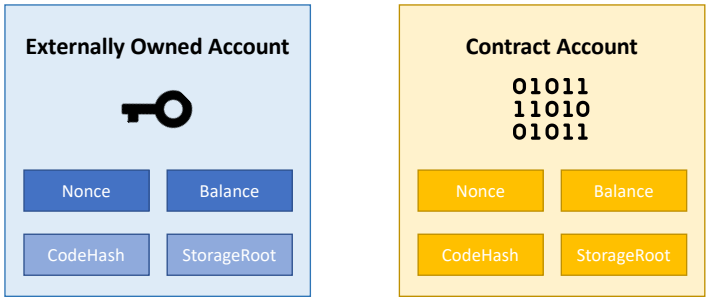



**Figure 2.4:** Anatomy of Externally Owned Accounts (EOAs) and Contract Accounts (CAs).

- **Nonce:** A number that acts as a simple counter which indicates the number of transactions sent from the account. This prevents the replaying of transactions and ensures that transactions are only processed once. If the account is an EOA, this number represents the number of transactions sent from the EOA's address. If the account is a CA, the nonce represents the number of contracts created by the CA.
- **Balance:** The amount of cryptocurrency owned by this address in units of wei. Wei is the smallest subunit of ether (1 wei is equivalent to $10^{-18}$ ether).
- **StorageRoot:** A hash of the root node of the Merkle Patricia tree that encodes the storage contents of the account. This value is by default empty for both types of accounts and is solely updated for CAs whenever data is written to storage.
- **CodeHash:** A hash of the bytecode associated with this account. For CAs, this represents a hash of the code that is stored on the blockchain. For EOAs, the this value is the hash of the empty string.

Creating an EOA has no cost as no data such as code, storage or balance is associated with the account at creation time. A CA on the other hand, has a cost because it uses the blockchain's storage to persist the contract code and data directly at creation time.

### 2.1.3 Transactions

Transactions are essentially cryptographically signed instructions from EOAs to update the state of the Ethereum blockchain. EOAs sign their transactions using their private key in order to cryptographically prove that the transaction could only have come from them and not from someone else. Two types of transactions exist: message calls and contract creations. The latter are transactions with an empty recipient field that result in creating new CAs (i.e., smart contracts). The code, to be associated with the CA, is placed inside the data field of the transaction. Regardless of their type, all transactions contain the following fields (see Figure 2.5):

- **Nonce:** A count on the number of transactions sent by the sender. This number is incremented by one every time a transaction is sent by the sender.

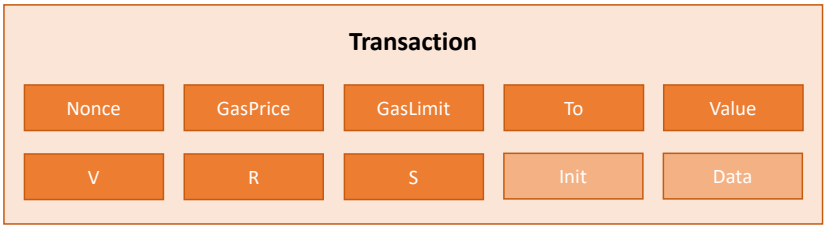



**Figure 2.5:** Anatomy of an Ethereum transaction.

- **GasPrice:** The amount of wei that the sender is willing to pay for each unit of gas that is used during the execution of the transaction.
- **GasLimit:** The maximum amount of gas units that the sender is willing to spent for the execution of this transaction. This amount is set and paid upfront, before any computation is performed.
- **To:** The address of the recipient. If the recipient is an EOA, the transaction will transfer value, if the recipient is a CA, the transaction will transfer value as well as execute the contract's code. A transaction with an empty recipient address is used to trigger the creation of a new CA.
- **Value:** The amount of wei to be sent from the sender account to the recipient account. Interestingly, this value may be used to set the starting balance of the newly created CA in a contract-creating transaction.
- **V, R, S:** These values represent the digital signature (R, S) which can be used to recover the public key (V). These values identify the sender of the transaction and confirms that the sender has authorized the transaction.
- **Init:** This field is only part of contract-creating transactions and consists of an unlimited length byte array that includes the code to be used during the initialization process and the code to be permanently associated with the newly created CA.
- **Data:** This is an optional field that is only part of message calls and consists of a byte array of unlimited size that specifies the input data (e.g., , function name, function parameters) of the message call.

As previously mentioned, contract-creating transactions and message calls are always initiated by EOAs. However, this does not mean that CAs cannot communicate with other CAs. CAs can send messages or so-called "internal transactions" to other CAs. Internal transactions are similar to normal transactions, with the major difference being that they are not initiated by EOAs, but instead they are initiated by CAs. Moreover, internal transactions merely exist as virtual objects that, unlike transactions, are not persisted in the Ethereum

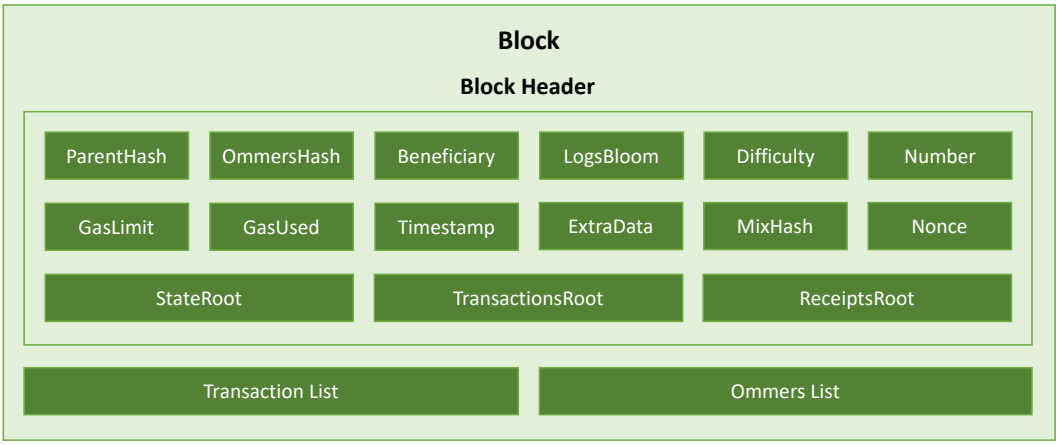



**Figure 2.6:** Anatomy of an Ethereum block.

blockchain and only exist at execution time. When a CA sends an internal transaction to another CA, the code that is associated with the recipient CA is executed. In contrast to normal transaction, internal transactions do not contain a gas limit by default. They are only limited by the gas limit that was determined by the normal transaction that triggered them. Thus, the gas limit that the EOA provides within its normal transaction, must be large enough to perform the execution of the normal transaction, including any sub-executions that occur as a result of that transaction, such as any internal transactions. If the execution of an internal transaction runs out of gas, then its execution will be will be reverted, along with any subsequent internal transactions triggered by the execution. However, the parent execution is not reverted.

### 2.1.4 Blocks

Blocks are batches of transactions with a reference to the hash of the previous block. This adds immutability and makes fraud noticeable, since a change in a transaction would invalidate all the previous blocks as all previous block hashes would change as well. Moreover, by grouping transactions into blocks, all network participants are given enough time to come to consensus, even in the case where hundreds of transactions are broadcast per second. The size of a block is usually bounded to a target size of 15 million gas units. However, the size of blocks will be increased or decreased depending on the demands of the network, up to a block limit of 30 million gas units (2x target block size). The total amount of gas spent by all transactions contained within the block must be less than the block gas limit. This is crucial as otherwise blocks could grow arbitrarily large and congest the blockchain. A block consists of a block header, a list of transactions, and a list of the block headers of the block's ommers (see Figure 2.6). A block header consists of the following fields:

- **ParentHash:** A hash of the parent (previous) block's header (i.e., the pointer that links blocks together in a chain).

- **OmmersHash:** A hash of the current block's list of ommers. An ommer is a block whose parent is equal to one of the current block's parent's parent.
- **Beneficiary:** The account address that receives the fees for mining this block.
- **LogsBloom:** A Bloom filter (i.e., a data structure) that allows efficient querying of information contained in the logs.
- **Difficulty:** The effort required to mine this block.
- **Number:** The count of the current block. The block number of the genesis block starts at number zero and each subsequent block number is increased by one. The block number is often referred as the length of the blockchain in blocks.
- **GasLimit:** The current gas limit per block. This value represents the limit set on the overall gas consumption for this block.
- **GasUsed:** The sum of the total gas used by all transactions contained in this block. This value cannot surpass the GasLimit.
- **Timestamp:** The UNIX timestamp when the block was mined.
- **ExtraData:** Arbitrary data that can set by the miner. This data is limited to 32-byte and usually refers to the name of the miner or the client version that was used to mine the block.
- **MixHash:** A hash that, when combined with the nonce, proves that this block meets the difficulty of this block.
- **Nonce:** A value that, when combined with the MixHash, proves that this block has performed the required work.
- **StateRoot:** The hash of the root node of the Merkle Patricia trie that stores the state of all accounts (i.e., account balances, storage, code, and nonces). The hash is calculated only after all transactions have been executed.
- **TransactionsRoot:** The hash of the root node of the Merkle Patricia trie that stores all transactions listed in this block.
- **ReceiptsRoot:** The hash of the root node of the Merkle Patricia trie that stores the receipts of all transactions listed in this block. Transaction receipts are generated after the execution of a transaction contain information such as logs or the actual gas that has been used during execution.

Block times are much lower in Ethereum (~15 seconds) than compared to those of other blockchains, such as Bitcoin (~10 minutes). The block time refers to the time that it takes to mine a new block. In Ethereum, the average block time is evaluated after each block. The expected block time is set as a constant at the protocol level and is used to protect the network's security when miners gain more computational power. The average block time is compared with the expected block time, and if the average block time is higher, then the difficulty contained in the block header is decreased. If the average block time is smaller, then the difficulty in the block header is increased. A smaller average block time enables faster transaction processing. However, shorter block times also means that miners will be more likely to find more competing block solutions. These competing blocks are often referred as ommers or uncles and are seen as "orphaned blocks", hence, blocks that were mined but are not part of the canonical chain. The purpose of ommers is to incentivise miners to include orphaned blocks as a part of the canonical chain and thereby avoid forks. Miners are only allowed to include orphaned blocks that are not more than six block numbers away from the current block number.

### 2.1.5 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is a purely stack-based, register-less virtual machine that runs low-level bytecode and supports a Turing-complete set of instructions. Every instruction is represented by a one-byte opcode (e.g., `0x60` $\rightarrow$ `PUSH1`, e.g., `0x50` $\rightarrow$ `POP`, etc.). The instruction set consists of more than 140 different instructions ranging from basic operations such as arithmetic operations or control-flow operations to more specific ones, such as the modification of a contract's storage or the querying of properties related to the executing transaction (e.g., sender) or the current blockchain state (e.g., block number). The EVM uses a memory model that is specific to the execution of smart contracts and differs from the traditional von Neumann architecture (see Figure 2.7).

Instead of organizing code and data in one large general-purpose memory, the EVM follows the Harvard architecture by separating code and data into four different address spaces: (1) an immutable code address space, which contains the smart contract's bytecode, (2) a mutable but persistent storage address space that allows smart contracts to persist their data across executions, (3) a mutable but volatile memory address space that acts as a temporary data storage for smart contracts during execution, and finally (4) a stack address space that allows smart contracts to pass arguments to instructions at runtime.

The EVM employs a gas mechanism that assigns a cost to the execution of an instruction. This mechanism prevents DoS attacks and ensures termination by solving the halting problem. When issuing a transaction, the sender has to specify a gas limit and a gas price. The gas price defines the amount of ether that the sender is willing to pay per unit of gas used. As the gas price is coupled to ether, developers are motivated to write efficient pro-

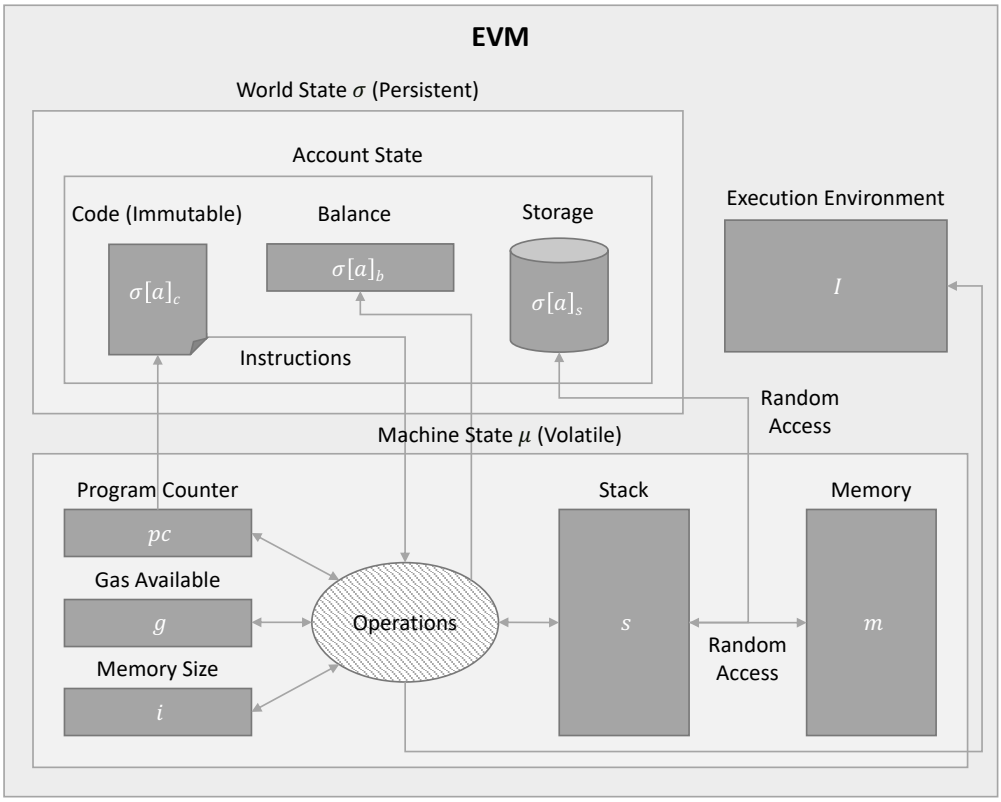



**Figure 2.7:** Architecture of the Ethereum Virtual Machine.

grams to keep transaction costs low and to avoid "infinite" loops. The base fee for executing a transaction starts at 21,000 gas units. The final execution costs are determined by multiplying the gas price with the gas used. The gas limit is specified in gas units and must be large enough to cover the amount of gas consumed by the instructions during a contract's execution, otherwise execution will terminate abnormally with an out-of-gas exception and its effects will be rolled back. The EVM also throws an exception when a jump destination is invalid, an instruction does not exist, or when there are not enough elements on the stack to perform an given operation.

Instructions operate on a *stack* of 256-bit big-endian words. The stack is private to a single contract (but not to methods within the contract) and is almost free to use in terms of gas. The size of the stack is limited to 1,024 items. In addition to the stack, smart contracts can also store variables in *memory*. Memory is a random-access array of 256-bit words that is accessible only by the currently executing smart contract. Memory is always initialized with zeros and thus isolated from previous executions. Memory is also used to pass arguments across message calls. Figure 2.8 shows an example of an EVM message call. The `CALL` instruction first copies the arguments of the message call from memory into the input data of a new instance of the EVM. The control is then returned back to the message caller via the `RETURN` instruction and the return value that includes the result of the message call is placed into the memory of the caller. Besides stack and memory, the EVM also features *storage*.

While stack and memory are volatile and may only hold values during execution, storage is persistent and part of the world state $\sigma$. It is organized as a Patricia Merkle trie holding sets of key-value stores for each account. Storage is isolated from other smart contracts and is the only way for a smart contract to retain state across executions. While storage is theoretically unlimited, its use is expensive (compared to stack and memory) and should only be used to store small amounts of data.

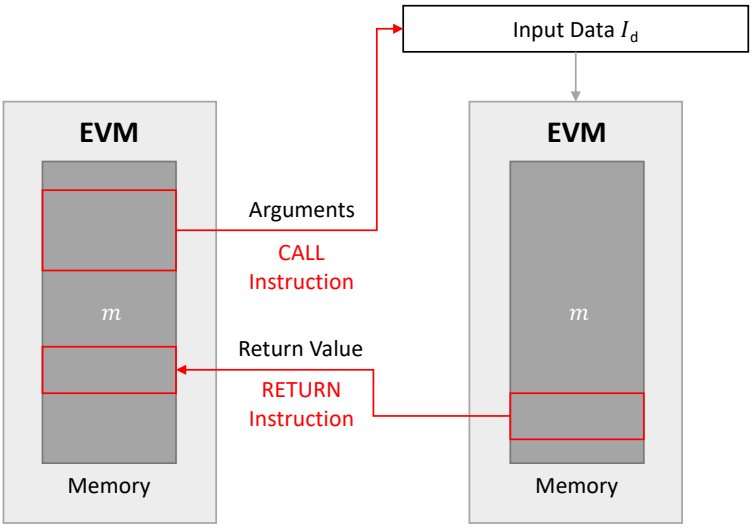



**Figure 2.8:** EVM message call.

During execution, the EVM holds a machine state $\mu = (g, pc, m, i, s)$, where $g$ is the gas available, $pc$ is the current program counter, $m$ represents the memory contents, $i$ is the active number of words in memory, and $s$ is the content of the stack. The execution of a smart contract results in the modification of the world state $\sigma$, which is a data structure that holds a mapping of an address $a$ to an account state $\sigma[a]$. An account state contains the bytecode $\sigma[a]_c$, a balance $\sigma[a]_b$, and a storage $\sigma[a]_s$. Apart from the world state $\sigma$, the EVM also has access to the execution environment $I = (I_a, I_o, I_p, I_d, I_s, I_v, I_b, I_H, I_e, I_w)$, where $I_a$ is the address of the account that is being executed, $I_o$ is the transaction origin, $I_p$ is the gas price, $I_d$ is the transaction input data, $I_s$ is the transaction sender, $I_v$ is the transaction value, $I_H$ is the current block header information, $I_e$ is the current call depth, and $I_w$ is the permission to make modifications to the world state. In contrast to traditional programs, the effect of a smart contract execution may only depend on information originating from the world state $\sigma$ and the execution environment $I$. This guarantees reproducibility, which enables determinism and therefore allows the Ethereum blockchain to reach a consensus. In summary, the EVM essentially takes as input the current world state $\sigma$ and an execution environment $I$, and outputs a updated world state $\sigma'$, which is afterwards used as input for the next execution.

## 2.2 Smart Contracts

The concept of smart contracts has been first introduced by Nick Szabo in 1997 [6]. He described the concept of a trustless system consisting of self-executing computer programs that would facilitate the digital verification and enforcement of contract clauses contained in legal contracts. Due to the lack of efficient trustless systems, the concept of smart contracts remained out of reach for many years, and it only became a reality with the release of blockchains, such as Ethereum.

### 2.2.1 Solidity

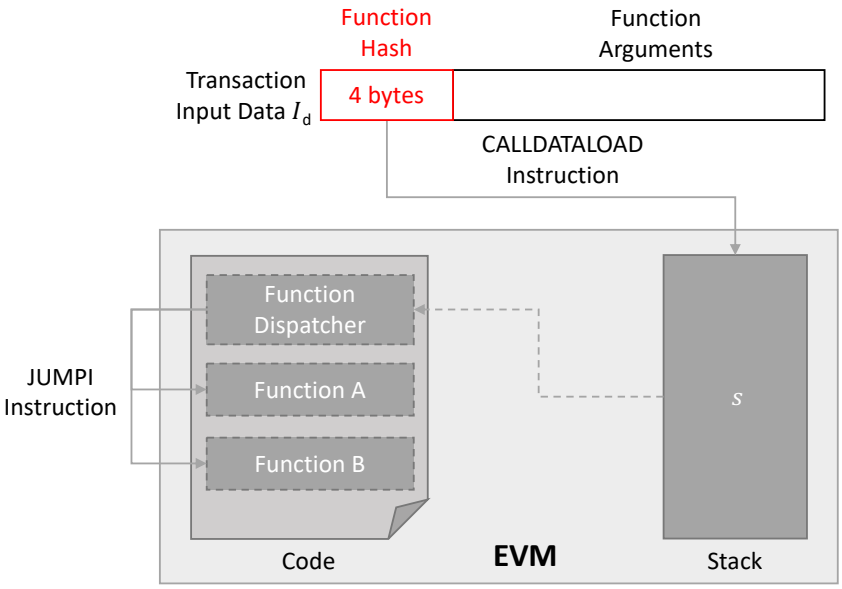



**Figure 2.9:** Solidity function dispatcher.

Solidity is an object-oriented programming language and it is currently the most prominent programming language for developing smart contracts in Ethereum. Its syntax resembles a mixture of C and JavaScript, but it comes with a variety of unique concepts that are specific to smart contracts and might be unfamiliar or confusing for new developers, such as the visibility of function modifiers: `internal`, `external`, `pure`, `view`, the function-wide scoping of variables, the emitting of events, or smart contract specific operations such as `selfdestruct` or `revert`. Similar to C, Solidity uses a function dispatch table to select what function to execute during runtime. The compiler does so by adding to the bytecode a function dispatcher that first loads the hash of the name of the function to be executed (the initial 4 bytes of the transaction input data $I_d$) into the stack and then jumps to the function implementation that is associated with the hash (see Figure 2.9). Now, unlike C and JavaScript, the concept of “undefined” or “null” values does not exist in Solidity. Newly declared variables always

**Table 2.2:** Functions that can move ether in Solidity.

| | Adjustable Gas | Gas Limit | Behavior on Error |
|---|---|---|---|
| `send()` | No | 2,300 | Return False |
| `transfer()` | No | 2,300 | Throw Exception |
| `call.value()` | Yes | All/Settable | Return False |

have a default value depending on their type. For example, integers are always initialized with zero, whereas Boolean's are always initialized with `False`. Besides common variable types, such as `int`, `bool`, or `string`, Solidity also provides unique types such as `mapping` or `address`. The variable type `mapping` behaves similar to a dictionary in Python by mapping keys to values and therefore allowing an easy access to storage. The variable type `address` is meant to hold addresses of Ethereum accounts and has member function such as `balance`, `send`, `transfer`, `call`, etc. The member functions `send`, `transfer`, and `call` can be used to move ether from one address to another. All these functions make use of the EVM `CALL` instruction, with difference being the gas limit that can be used and the behavior on an error (see Table 2.2 for more details). For example, the functions `send` and `transfer` have a fixed gas limit of 2,300 units, whereas the function `call` by default provides all the available gas to the `CALL` instruction, unless the developer explicitly states the gas limit by using the function member `gas` (e.g., `call.value().gas()`).

Almost all variable types in Solidity are of type value, meaning that their value is copied when passed as an argument to a function. In contrast, the value of variable types of type reference, are not copied and therefore modified across function calls. Currently, variable types of type reference comprise mappings, arrays, and structs. Developers always have to explicitly state the data area where the type is stored when using a variable of type reference: `memory` (where its lifetime is limited to an external function call), or `storage` (where its lifetime is limited to the lifetime of the contract). In general, state variables are always stored in storage, function arguments are always stored in memory, and local variables of type value are stored in the stack. Moreover, Solidity suggests to be a statically typed language, i.e., the compiler expects type information for each variable to be made explicit. For instance, integers can be signed and unsigned, and of lengths between 8 and 256 bits in 8-bit steps denoted as `uint8` or `int128`. This resembles integer types in C and may lead novice developers to assume that a `uint8` will occupy 8 bits in memory, while an `int128` occupies 128 bits. However, this assumption is wrong. Any integer type will be represented inside the EVM as 256-bit values in big endian order using two's-complement. That is, the integer type system of Solidity is not entirely consistent with that of the EVM, which can lead to programming errors. Explicit conversion between primitive types is possible, but the effects are not well documented. In fact, the documentation reads: *Note that this may give you some unexpected behavior so be sure to test to ensure that the result is what you want*.

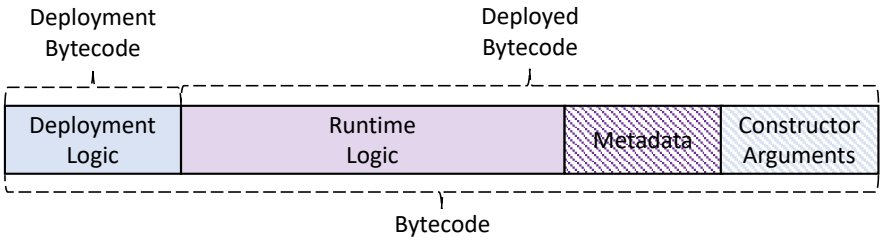



**Figure 2.10:** An illustrative example of the anatomy of Ethereum bytecode.

For example, explicitly casting a signed negative integer into an unsigned one will not result in the absolute value, but rather simply leave its bit-level representation intact. Interestingly, Solidity does not support floating points. However, similar to JavaScript, Solidity provides members such as `length` or `push` for arrays.

Apart from variables, Solidity also provides units for ether and time (e.g., `wei`, `days`, etc.) as well as special keywords and functions which always exist in the global namespace and are mainly used as general-use utility functions or to provide information about the blockchain. For instance, the keyword `block` can be used to access information about the current block (e.g., `block.number`), and the keyword `msg` and `tx` to access information about the transaction/message that is being executed (e.g., `tx.origin`, `msg.sender`, etc.). Also mathematical and cryptographic functions are provided (e.g., `addmod`, `ecrecover`, etc.). For error handling, Solidity provides two convenience functions called `assert` and `require`, that can be used to check for a condition and to throw an exception if the condition is not met. Developers can interleave Solidity statements with inline assembly in a language called Yul, which is close to the one of the EVM. This gives more fine-grained control and bypasses optimizations imposed by the compiler. An inline assembly block is marked via the keyword `assembly`. The inline assembly code can access local Solidity variables, but different inline assembly blocks cannot call functions or access variables defined in a different inline assembly block.

### 2.2.2 Bytecode

Ethereum bytecode consists of a sequence of bytes that is interpreted by the EVM. Each byte either encodes an instruction or data. Figure 2.10 depicts the anatomy of Ethereum bytecode. Ethereum bytecode consists of two main parts: *deployment bytecode* and *deployed bytecode*. Deployment bytecode includes the deployment logic of the smart contract. This logic is responsible for initializing state variables and reading constructor arguments appended at the end of the Ethereum bytecode. It is also in charge of extracting the deployed bytecode from the Ethereum bytecode and copying it to persistent storage. This is achieved via the `CODECOPY` and `RETURN` instructions. Starting from a given offset and for a given size, the `CODECOPY` instruction first copies the code running in the current environment to memory.

Afterwards, the RETURN instruction returns the code that has been copied into memory to the EVM. As a result, the EVM creates a new contract by generating a new 160-bit address and persisting the returned code with this new address. The deployed bytecode contains the *runtime logic* (i.e., runtime bytecode) and optional *metadata*. The runtime bytecode is the logic that is executed whenever a transaction is sent to a smart contract. Some compilers, such as the Solidity compiler, also append some metadata (e.g., compiler version) to the end of the runtime bytecode.

### 2.2.3 Vulnerabilities

Atzei et al. [60] were the first to provide a survey of smart contract vulnerabilities and attacks. However, their survey relates to 2016. Since then, several new vulnerabilities and attacks have emerged (e.g., Parity wallet hacks, integer overflows, etc.). In 2018, the NCC Group released their Top 10 ranking of smart contract vulnerabilities and dubbed it the Decentralized Application Security Project (DASP)[1]. DASP is an open and collaborative project to join efforts in documenting and ranking smart contract vulnerabilities within the security community. The idea is similar to the well known Open Web Application Security Project (OWASP). However, neither the ranking nor the list of vulnerabilities has been updated since its first release in 2018. Another initiative to document well-known attacks is the one presented by ConsenSys [38]. Unfortunately, their list only contains a handful of vulnerabilities. In 2019, Chen et al. [145] presented a survey of vulnerabilities, attacks, and defenses for the Ethereum blockchain. Moreover, a project called the Smart Contract Weakness Classification Registry (or simply SWC Registry[2]) has been released with the goal to provide a common way to report and classify security issues in smart contracts. It loosely resembles the terminology and structure used in the Common Weakness Enumeration (CWE) project. At the time of writing, it includes 37 entries. Each entry consists of an identifier (SWC-ID), a weakness title, a CWE parent, and a list of related code samples. Unfortunately, Chen et al.'s survey as well as the SWC Registry lack both a ranking and labels which highlight the severity and impact of each vulnerability.

Table 2.3 depicts our own taxonomy of vulnerabilities and detection tools. It lists 22 different vulnerabilities, grouped into 10 categories, and mapped to 25 distinct vulnerability detection tools: 15 tools based on static analysis and 10 on dynamic analysis. The tools listed in our taxonomy were chosen based on the following two criteria: (1) the tool does not just construct artifacts such as control-flow graphs or intermediate representations of smart contracts but also detects vulnerabilities; (2) the tool identifies at least one vulnerability listed in the DASP ranking. Interestingly, when creating this taxonomy, we realized that there is no unified naming or definition of vulnerabilities. We found that sometimes tools detect the same vulnerability, but name the vulnerability differently. For example, while VANDAL uses

[1] https://dasp.co
[2] https://swcregistry.io

**Table 2.3:** A taxonomy of vulnerabilities and detection tools. Vulnerabilities are sorted according to the DASP ranking. Tools marked with ● detect the vulnerability, while tools marked with ○ do not detect the vulnerability. Tools that only partially detect a vulnerability or just some vulnerabilities of a category are marked with ◐. Tools with available source code are marked with ∗, whereas tools without available source code are marked with †.

| | Static | | | | | | | | | | | | | | | | Dynamic | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Vulnerability** | ∗ OYENTE [76] | † ZEUS [112] | ∗ MAIAN [122] | ∗ MYTHRIL [120] | ∗ SECURIFY [136] | ∗ OSIRIS [102] | ∗ MANTICORE [87] | ∗ SLITHER [155] | ∗ SMARTCHECK [135] | ∗ VANDAL [90] | ∗ MADMAX [103] | † SMARTSCOPY [156] | ∗ TEETHER [116] | † ETHAINTER [174] | ∗ ETHBMC [181] | ∗ ECFCHECKER [69] | † SEREUM [165] | ∗ CONTRACTFUZZER [110] | ∗ ETHRACER [115] | ∗ VULTRON [169] | † HARVEY [171] | ∗ ILF [160] | ∗ ÆGIS [180] | † SODA [175] | ∗ CONFUZZIUS [210] |
| **Reentrancy** | ◐ | ◐ | ○ | ◐ | ◐ | ○ | ◐ | ◐ | ◐ | ◐ | ○ | ◐ | ○ | ○ | ○ | ◐ | ◐ | ◐ | ○ | ◐ | ◐ | ○ | ◐ | ◐ | ◐ |
| *Same-Function* | ● | ● | ○ | ● | ● | ○ | ● | ● | ● | ● | ○ | ● | ○ | ○ | ○ | ● | ● | ● | ○ | ● | ● | ○ | ● | ● | ● |
| *Cross-Function* | ○ | ○ | ○ | ● | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ |
| *Delegated* | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ |
| *Create-Based* | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ |
| *ERC777-Based* | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| **Access Control** | ○ | ◐ | ○ | ● | ● | ○ | ◐ | ◐ | ◐ | ◐ | ○ | ◐ | ◐ | ◐ | ◐ | ○ | ○ | ◐ | ○ | ○ | ○ | ◐ | ◐ | ○ | ◐ |
| *Transaction Origin* | ○ | ● | ○ | ● | ● | ○ | ● | ● | ● | ● | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ |
| *Unchecked Delegatecall* | ○ | ○ | ○ | ● | ● | ○ | ● | ● | ○ | ○ | ○ | ● | ● | ● | ● | ○ | ○ | ● | ○ | ○ | ○ | ● | ○ | ○ | ● |
| *Parity Wallet Hack 1* | ○ | ○ | ● | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ○ | ● |
| *Unchecked Selfdestruct* | ● | ○ | ● | ● | ● | ○ | ● | ● | ○ | ● | ○ | ● | ● | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ● |
| **Arithmetic** | ● | ● | ○ | ● | ○ | ● | ● | ○ | ◐ | ○ | ◐ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ● | ○ | ● |
| *Integer Overflow* | ● | ● | ○ | ● | ○ | ● | ● | ○ | ◐ | ○ | ◐ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ● | ○ | ● |
| *Integer Underflow* | ● | ● | ○ | ● | ○ | ● | ● | ○ | ◐ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ● | ○ | ● |
| **Unhandled Exception** | ● | ● | ○ | ● | ● | ○ | ● | ● | ● | ● | ○ | ● | ○ | ○ | ○ | ○ | ○ | ● | ○ | ● | ● | ● | ○ | ● | ● |
| **Denial-of-Service** | ○ | ◐ | ◐ | ◐ | ◐ | ○ | ○ | ○ | ◐ | ○ | ◐ | ◐ | ◐ | ◐ | ◐ | ○ | ○ | ◐ | ○ | ◐ | ○ | ◐ | ◐ | ○ | ◐ |
| *Unexpected Throw* | ○ | ● | ○ | ● | ○ | ○ | ○ | ○ | ● | ○ | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| *Insufficient Gas Griefing* | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ○ | ○ | ○ | ● | ○ | ● | ○ | ○ | ○ | ○ | ○ |
| *Block Gas Limit* | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ● | ○ | ● | ● | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| *Block Stuffing* | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| *Parity Wallet Hack 2* | ○ | ○ | ● | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ○ | ● |
| **Bad Randomness** | ◐ | ● | ○ | ● | ● | ○ | ● | ○ | ◐ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● | ○ | ◐ | ● |
| **Frontrunning** | ● | ● | ○ | ● | ● | ○ | ● | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● | ○ | ● |
| **Time Manipulation** | ● | ● | ○ | ● | ● | ○ | ● | ● | ● | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ● | ● |
| **Short Address** | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ |

the name *unchecked send* for an *unhandled exception*, OYENTE names this vulnerability *mishandled exception*. These different nomenclatures make it hard to compare the detection capabilities of the individual tools. We extracted the detection capabilities of each tool by manually analyzing their source code if available. Otherwise, we inferred the detection capabilities directly from their paper. MYTHRIL and MANTICORE have frequent release schedules: the capabilities highlighted in Table 2.3 are based on their latest version, which at the time of writing were, 0.22.4 and 0.3.3, respectively. It is interesting to observe that there is no single tool that is capable of detecting all the vulnerabilities listed in Table 2.3. We provide in the following a detailed explanation of each of the vulnerabilities.

**Reentrancy.** A typical flaw in smart contracts are reentrant calls. Reentrancy may occur whenever a contract sends a value to or calls a function from another contract, and the called contract has enough gas to call back the original contract. As a result, the called

contract reenters the original contract. Therefore, the original contract must guarantee that its state is always correct, independent of reentrant calls. Otherwise, a malicious contract can take advantage of an incorrect intermediate state to steal funds. The 2016 DAO hack remains the most famous example of a reentrancy attack not only because an unknown attacker managed to steal funds worth 50M USD [47], but also because it led to a hard fork of the Ethereum blockchain [163]. A way to protect against reentrancy is to use the Solidity methods `send` or `transfer` rather than `call`. The aforementioned methods limit the amount of gas to 2,300 units, which makes it impossible for the called contract to modify the state or make any function calls. However, this only protects the sending of value.

In 2018, an attacker exploited a reentrancy bug by making use of an unprotected function call [125]. Whenever a contract calls an external function that belongs to another contract, the amount of gas will not be limited. The only way to protect against reentrancy, in that case, is to add a reentrancy guard in the form of a mutex. Reentrancy vulnerabilities were extensively studied by Rodler et al. [165], and can be divided into four distinct categories: *same-function*, *cross-function*, *delegated* and *create-based*. Same-function reentrancy occurs when an attacker reenters the original contract via the same function, whereas cross-function reentrancy occurs when an attacker takes advantage of another function that shares the same state with the original function. Delegated reentrancy and create-based reentrancy are similar to same-function reentrancy, but use different opcodes to perform reentrancy. While delegated reentrancy makes use of `DELEGATECALL` or `CALLCODE`, create-based reentrancy uses `CREATE` or `CREATE2`.

Despite reentrancy having been studied so well, it seems to remain a reoccurring problem in smart contracts. In early 2019, the scheduled Ethereum Constantinople hard fork introduced a cheaper gas cost for certain state-changing operations. As an unwanted side effect, this enabled reentrancy attacks, even when using safe methods such as `send` or `transfer` [144]. Fortunately, the issue has been reported on time, and the hard fork has been postponed, thereby preventing it from affecting any smart contracts on the blockchain. However, in April 2020, two decentralized exchanges, Uniswap and Lendf.me, became victims of a reentrancy attack due to them trading the imBTC token [187]. This particular token implements the ERC777 standard [74], which allows calling another contract before tokens are transferred and therefore enabling reentrancy. The issue had already been reported a year before the attack [167, 164]. However, many developers assumed that this vulnerability would only affect the transfer of ether and that the transfer of tokens would be safe.

**Access Control.** A typical design pattern in smart contracts is to assign an address as the owner of the smart contract [106]. This address has then privileged access to functions that can either update sensible variables, transfer funds, or destroy the entire contract. Unfortunately, this also means that faulty implementations of access control can lead to devastating consequences. One example of a flawed access control implementation is the use of Solid-

ity's `tx.origin` to check whether the current address is authenticated to perform a sensible function call such as the withdrawal of funds. However, `tx.origin` does not represent the currently calling address, but the very first address that initiated the transaction. Remember that within a transaction, contracts can call other contracts, and therefore the calling address can be a different one at runtime. An attacker can perform a man-in-the-middle attack by first convincing a victim to send a transaction to the attacker's contract, which then performs within the same transaction a message call to the victim's wallet. If the wallet checks for the transaction origin to authenticate users, then the attacker will be authenticated as the victim and might be able to steal the funds. Developers should, therefore, rely on the `msg.sender` property to authenticate user accounts.

A different typical design pattern is the proxy contract. It separates funds and data from logic. The idea of proxy contracts was introduced to mitigate the problem that smart contracts, once deployed, cannot be modified. Proxy contracts are implemented using Solidity's `delegatecall` method. This method takes an address as parameter and executes code residing at that address in the context of the original contract. If an attacker can manipulate the address to his own will, then he or she can execute arbitrary code, including code that steals funds or destroys the original contract. This vulnerability is known as *unchecked delegatecall* or *unsafe delegatecall*. A prime example of such a vulnerability is the first Parity wallet hack back in July 2017 [84]. The wallet contract contained code that would redirect any unmatched function calls to a library contract using a `delegatecall`. Unfortunately, the developers forgot to write a check for the `initWallet` function, ensuring that the function could only be called once. As a result, an attacker was able to gain ownership of the contract by simply forwarding the call to that function through the wallet contract. Once in control, the attacker withdrew all the funds by invoking another function called `execute`. This vulnerability resulted in an estimated loss of roughly 150K ether, approximately 30M USD at that time.

Another popular design pattern is the implementation of a `kill` or `destroy` function that allows the owner to remove the code of the smart contract from the blockchain once it is not needed anymore. Such functions are implemented using Solidity's `selfdestruct` method. This method deletes the contract's code and sends all the funds to a given address. However, if access to this method is not adequately protected, then an attacker can either destroy the contract or steal its funds. This vulnerability is known as *unchecked selfdestruct* or *suicidal contract*. A prominent example of this vulnerability is the second Parity wallet hack (see *denial-of-service* vulnerability below).

**Arithmetic.** Integer overflows and underflows are not specific to smart contracts, but they are well-known vulnerabilities in software engineering. However, they are especially dangerous in smart contracts because they can quickly lead to exploits that allow attackers to steal large amounts of funds. An integer overflow or underflow occurs when the result of an arithmetic operation falls outside of the range of an integer type. For example, an overflow

occurs if a variable of type `uint256` contains its maximum value (i.e., $2^{256} - 1$) and a value of one is added to the variable. The value of the variable circles then back to zero. The same is true for underflows. If an unsigned variable contains zero and a value of one is subtracted, then the variable will be set to its maximum value. Integer overflow attacks were first reported in conjunction with the two ERC20 token contracts called BeautyChain [126] and MESH [127]. Both attacks are a result of improper input validation on numeric inputs. Since neither the Solidity compiler nor the EVM enforces integer overflow/underflow detection, a common way to mitigate integer overflows and underflows is to use the `SafeMath` library for arithmetic operations [124].

**Unhandled Exception.** This vulnerability is sometimes also described as *exception disorder*, *unchecked send*, or *unchecked low-level call*. The reason for this vulnerability is that Solidity does not handle exceptions uniformly. For example, methods such as `call`, `delegatecall` or `send`, do not stop the execution or revert the entire transaction upon failure. Instead, an exception is raised and propagated up to the method, reverting only the side effects caused by the method call. The execution is then resumed with the method returning `false`. Beyond that point, it is the responsibility of the developer to check the result of the method and to perform appropriate exception handling. However, in practice, many developers forget or decide to ignore the handling of such exceptions. As a result, an attacker can create a contract that intentionally causes the method call to fail, for example, either by throwing an exception or by consuming all the gas (see the *denial-of-service* vulnerability below). An example of an unhandled exception is the King of the Ether Throne (KotET) incident in February 2016 [46]. The contract failed to process a legitimate payment because the recipient was a contract, and the amount of 2,300 gas units provided by `send` was not enough. The payment failed, and the ether was returned to the KotET contract. The contract code did not check for payment failure and continued processing, making the latest player king, despite the compensation payment not having been sent to the previous player.

**Denial-of-Service.** There are many ways to cause a denial-of-service (DoS) on a smart contract. These can range from artificially increasing the gas consumption of a function, to abusing faulty access control in order to destroy a smart contract. Gas plays a crucial role in smart contracts. If a user does not provide enough gas or if a function consumes too much gas, then this can result in an exception. If unanticipated, then this exception can result in a smart contract being in an inconsistent state with the consequence of locking or freezing up funds. This type of DoS is known as *insufficient gas griefing* or *gasless send*. Another type of DoS can be due to unbounded operations. In April 2016, a smart contract called GovernMental was stuck in a deadlock with a balance of 1,100 ether, because the list of creditors was so long, that it would require an amount of gas that is higher than the allowed maximum to payout all the creditors in one transaction [40].

However, even if a contract does not contain unbounded operations, an attacker can still prevent other transactions from being included in the blockchain for several blocks. An attacker can continuously issue transactions, which will consume the entire block gas limit, such that no other transactions will be included in the block (given a high enough gas price). This type of DoS is known as *block stuffing* and can also be seen as a form of frontrunning (see *frontrunning* vulnerability below). A block stuffing attack was conducted in October 2018 on Fomo3D, a gambling contract that was designed to reward the last address that purchased a lottery ticket. Each purchase extended the timer, and the game ended once the timer went to zero. The attacker bought a ticket and then stuffed 13 blocks in a row until the timer was triggered and the payout was released to the attacker [123]. Besides gas-related DoS attacks, funds can also be stuck because the called contract throws on purposely an exception via a `revert`. This type of DoS is known as *unexpected throw*.

The second Parity wallet hack, which happened in November 2017 [80], can also be seen as a DoS attack. After the first Parity wallet hack, a new library contract was deployed in order to address the issues related to the first hack. However, the developers forgot to initialize the library contract after deployment, meaning the contract itself had no owners. As a result, three months after deployment, a user known as *devops199* was able to set itself as the owner and kill the contract via the `selfdestruct` method, which removed the code of the library contract from the blockchain [66]. The library contract itself contained no funds, but it was an inherent part of many Parity wallets. Consequently, any wallet trying to use the library contract failed. This effectively rendered the wallets unusable and resulted in freezing up the funds contained in the wallets for eternity.

**Bad Randomness.** Randomness is hard to achieve in general. In Ethereum it is even harder, since smart contracts are executed in a deterministic way. However, games and lotteries sometimes require non-deterministic values. Therefore, a variety of smart contracts rely on “unpredictable” information originating from yet unmined blocks, such as `blockhash` or `number`. They most often use this information as a seed to generate pseudo-random numbers. However, because the sources of randomness are predictable, malicious users can brute force the values of blocks before they have been mined. The Run [43] and SmartBillions [68] are famous examples of smart contract lotteries using block information for generating random numbers. In October 2017, an attacker was able to exploit the predictability of block information and steal 400 ether from the SmartBillions contract [83].

**Frontrunning.** Users observing the network can see and react to transactions before they are mined. Miners typically order transactions based on their gas prices. This results in gas price wars between users in the network. Frontrunning is therefore also known as *transaction order dependence* (TOD). A decentralized exchange is a perfect example on how this can be exploited. An attacker observes a transaction containing a large buy order and

quickly broadcasts its own transaction containing a buy order with a larger gas price in order to be executed before the observed buy order. A decentralized exchange named Bancor was reported in 2017 to be vulnerable to frontrunning attacks [82, 73]. A few other cases related to frontrunning have been studied by Eskandari et al. [151]. Common mitigations against frontrunning include the use of commit and reveal schemes or the specification of a minimum or maximum acceptable price range on a trade, thereby limiting price slippage.

**Time Manipulation.** Smart contracts sometimes rely on the current time to either be able to lock a token sale, or unlock funds within a payment channel or a game. In Solidity, the current timestamp is retrieved via `block.timestamp` or `now`. However, this value can be set by miners and can therefore be manipulated. If a miner holds enough value inside a smart contract, then the miner could gain an advantage by choosing a suitable timestamp for a block that he or she mines. Smart contracts should, therefore, avoid relying on timestamps originating from blocks. Atzei et al. stated that the ponzi smart contract GovernMental is vulnerable to a timestamp manipulation attack [60]. However, there is no reported incident of such an attack in the wild.

**Short Address.** Function arguments are encoded as chunks of 32 bytes within the `input` field of a transaction. However, if the length of an encoded argument is shorter than 32 bytes, then EVM will auto-pad extra zeros to the end of the argument such that it has a length of 32 bytes. Short address attacks have been first described by the authors of the Golem project [79]. They noticed the bug because of a user transaction failing due to a considerable amount of tokens being transferred. The ERC20 function `transfer(address to, uint256 value)` takes as input an address and an amount of tokens. A user entered an address that was shorter than 160-bit. Thus, the EVM added trailing zeros to the end of the transaction input, which resulted in shifting the value to the left by a few zeros and increasing the number of tokens to be transferred. Attackers can exploit this vulnerability by generating specially-crafted addresses that end with trailing zeros. Unfortunately, the EVM does not check the validity of addresses, and thus the only way to prevent this attack is to check the length of a transaction's input at the level of the smart contract using `msg.data` [78].

# Part I

# Vulnerabilities

# 3 | Osiris

### *Hunting for Integer Bugs in Smart Contracts*

*In this chapter, we focus on detecting integer bugs, a class of vulnerabilities that is particularly difficult to avoid due to certain characteristics of the EVM and the Solidity programming language. We start by introducing* OSIRIS *– a tool that combines symbolic execution and constraint solving in order to accurately find integer bugs in Ethereum smart contracts. Moreover, by employing taint analysis, we are capable of minimizing the number of reported false positives. We compare* OSIRIS *to existing tools and find that* OSIRIS *is capable of detecting a greater range of bugs, while providing a better specificity of its detection. Afterwards, we evaluate its performance on a large experimental dataset containing more than 1.2 million smart contracts. We find integer bugs in roughly 4% of the deployed contracts. Furthermore, we evaluate* OSIRIS *against a range of reported CWEs and identify critical vulnerabilities in a couple of token smart contracts that belong to the top 495 token smart contracts currently deployed on the Ethereum blockchain. Finally, we investigate causes for integer bugs and propose ways to protect smart contracts against integer bugs by suggesting modifications to the EVM and the Solidity compiler.*

## 3.1 Introduction

The capability of executing smart contracts is one of the most compelling features of modern blockchains. Smart contracts can carry assets worth millions of dollars. Unless explicitly designed by the developer, smart contracts typically cannot be changed once deployed on the blockchain. In that respect, it is imperative that smart contracts are correct and contain no vulnerabilities. Previous research identified a number of vulnerabilities specific to smart contracts (e.g., reentrancy), some of which led to prominent multi-million dollar fraud cases. However, little effort has been made in identifying classical software bugs, such as integer overflows, in smart contracts. Developers usually write their programs using a high-level programming language. The same applies for Ethereum smart contracts. In Ethereum, developers typically write their smart contract code in Solidity, which then compiles into EVM bytecode. Although various experimental versions of high-level languages exist, at the time of writing, Solidity [133] is the most prevalent language for developing smart contracts. At a

first glance, the C/JavaScript-like syntax of Solidity looks familiar to developers with experience in JavaScript or C, and encourages rapid development of smart contracts. However, the way how smart contracts are executed, as well as their security properties, are fundamentally different from traditional programs and may lead to unexpected behavior at runtime. In combination with a lack of strict static validation and limited support of development tools, developers are encouraged to tweak their smart contract code until it "just works". While this might be a feasible approach for prototyping, it likely results in fatal errors when smart contracts are deployed as real-world applications on the Ethereum blockchain. In contrast to traditional programs, once smart contracts have been deployed, they cannot be updated anymore. Transactions that were never intended by the developer become irreversible.

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to) {
4      credit[to] += msg.value;
5    }
6    function queryCredit(address to) returns (uint) {
7      return credit[to];
8    }
9    function withdraw(uint amount) {
10     if (credit[msg.sender] >= amount) {
11       msg.sender.call.value(amount)();
12       credit[msg.sender] -= amount;
13     }
14   }
15 }
```

**Figure 3.1:** A simplified version of the DAO smart contract.

With the DAO hack in June 2016 [57], it became clear what consequences emerge when subtle programming mistakes, in non-updatable smart contracts, hit high-volume applications. The attacker managed to drain 50 million USD worth of ether from the DAO smart contract, by exploiting a *"reentrancy"* bug in conjunction with a *"call to the unknown"* bug. Interestingly, Atzei et al. [60] reviewed the DAO hack one year later and realized that the attack could have been exploited more efficiently, using only two calls to the fallback function of the attacker's smart contract. The attack makes use of the previously reported vulnerabilities and a new unreported vulnerability: an integer underflow in the function `withdraw` at line 12 (see Figure 3.1). The attack works as follows. The attacker first deploys the contract `Mallory`. Afterwards, the attacker first invokes the function `attack` to donate 1 wei (1 wei $= 10^{-18}$ ether) to the DAO smart contract and then calls the function `withdraw` on the DAO smart contract to withdraw the funds (see Figure 3.2). The function `withdraw` will check whether the attacker has enough credit, and if yes, it will transfer the funds back to `Mallory`. As in the original attack, `call` will invoke `Mallory`'s fallback function (see line 11

```
1  contract Mallory {
2    SimpleDAO public dao = SimpleDAO(0x818EA...);
3    address owner;
4    bool performAttack = true;
5    function Mallory() {
6      owner = msg.sender;
7    }
8    function attack() {
9      dao.donate.value(1)(this);
10     dao.withdraw(1);
11   }
12   function() { // Fallback function
13     if (performAttack) {
14       performAttack = false;
15       dao.withdraw(1);
16     }
17   }
18   function getJackpot() {
19     dao.withdraw(dao.balance);
20     owner.send(this.balance);
21   }
22 }
```

**Figure 3.2:** A more efficient attack than the original DAO attack.

in Figure 3.1), which in turn will call back `withdraw`. This will interrupt the updating of the credit at line 12 in Figure 3.1. Hence, the check at line 10 in Figure 3.1 will succeed again, despite the attacker having already received all the funds it donated. Consequently, the DAO smart contract will send 1 wei to `Mallory` for the second time and invoke its fallback function again. However, this time the fallback function will do nothing and the nested calls will begin to close. The effect is that `Mallory`'s credit is updated twice: the first time to zero and the second time to $2^{256} - 1$ wei, because of an underflow occuring at line 12 in Figure 3.1. Now, to finalize the attack, the attacker simply calls the function `getJackpot`, which transfers all the funds from the DAO smart contract to `Mallory`.

In this chapter, we investigate the prevalence of such integer bugs in smart contracts. We introduce OSIRIS, a symbolic execution tool for detecting various types of integer bugs in Ethereum smart contracts. We use OSIRIS to find vulnerabilities in smart contracts currently deployed on the Ethereum blockchain. Furthermore, we investigate whether this type of vulnerability is already being exploited, and point out improvements to the EVM and the Solidity compiler as a safeguard against these types of bugs. In summary, this chapter makes the following contributions:

**Contributions**

- We present OSIRIS, a symbolic execution tool which automatically detects integer bugs in EVM bytecode. The tool currently covers three different types of integer bugs: *arithmetic bugs*, *truncation bugs* and *signedness bugs*.

- We run OSIRIS on all smart contracts that have been deployed on Ethereum until January 2018 (i.e., 1.2 million contracts), and find that 42,108 of them suffer from at least one of these three bugs.
- We compare OSIRIS to ZEUS [112] and demonstrate that OSIRIS is capable of detecting more vulnerabilities, with a higher confidence and a considerably lower false positive rate.
- We analyze 495 Ethereum top token smart contracts and discover major vulnerabilities in two them.
- We identify causes for integer bugs and propose modifications to the EVM and the Solidity compiler, to protect smart contracts against integer bugs.

## 3.2 Integer Bugs

A multitude of different scenarios exist where integer operations may result in bugs in smart contracts [44]. Three main classes of integer bugs exist that may allow a malicious user to steal ether or modify the execution flow of a smart contract: 1) *arithmetic bugs*, 2) *truncation bugs*, and 3) *signedness bugs*. All three classes of bugs occur due to the mismatch between machine arithmetic and arithmetic over unbounded integers.

**Table 3.1:** Behavior of integer operations in EVM and Solidity. Both $x$ and $y$ are $n$-bit integers, where $x_\infty$, $x_\infty$ denote their $\infty$-bit mathematical integers. Integer operations marked with $s$ denote signed operations, whereas integer operations marked $u$ denote unsigned operations.

| Integer Operation | In-Bounds Requirement | Out-of-Bound Behavior | |
|---|---|---|---|
| | | EVM [30] | Solidity [133] |
| $x +_s y$ | $x_\infty + y_\infty \in [-2^{n-1}, 2^{n-1}-1]$ | modulo $2^{256}$ | modulo $2^n$ |
| $x +_u y$ | $x_\infty + y_\infty \in [0, 2^n-1]$ | modulo $2^{256}$ | modulo $2^n$ |
| $x -_s y$ | $x_\infty - y_\infty \in [-2^{n-1}, 2^{n-1}-1]$ | modulo $2^{256}$ | modulo $2^n$ |
| $x -_u y$ | $x_\infty - y_\infty \in [0, 2^n-1]$ | modulo $2^{256}$ | modulo $2^n$ |
| $x \times_s y$ | $x_\infty \times y_\infty \in [-2^{n-1}, 2^{n-1}-1]$ | modulo $2^{256}$ | modulo $2^n$ |
| $x \times_u y$ | $x_\infty \times y_\infty \in [0, 2^n-1]$ | modulo $2^{256}$ | modulo $2^n$ |
| $x /_s y$ | $y \neq 0 \wedge (x \neq -2^{n-1} \vee y \neq -1)$ | $0$ if $y = 0$<br>$-2^{255}$ if $x = -2^{255} \wedge y = -1$ | $0^*$ / INVALID$^\dagger$ if $y = 0$<br>$-2^{n-1}$ if $x = -2^{n-1} \wedge y = -1$ |
| $x /_u y$ | $y \neq 0$ | $0$ | $0^*$ / INVALID$^\dagger$ |
| $x\ mod_s\ y$ | $y \neq 0$ | $0$ | $0^*$ / INVALID$^\dagger$ |
| $x\ mod_u\ y$ | $y \neq 0$ | $0$ | $0^*$ / INVALID$^\dagger$ |

$*$ Solidity version $<$ 0.4.0; † Solidity version $\geq$ 0.4.0

### 3.2.1 Arithmetic Bugs

We consider bugs such as integer overflows and underflows, but also bugs due to division by zero or modulo zero, as arithmetic bugs. Integer overflows (or underflows) occur when an

arithmetic expression results in a value that is larger (or smaller) than it can be represented by the resulting type. The usual behavior in such a case is to silently "wrap around", e.g., for a 32-bit type, reduce the value modulo $2^{32}$. In C/C++ the out-of-bounds behavior of integer operations is mostly undefined, whereas in Ethereum all behavior is well-defined. Table 3.1 summarizes the different out-of-bound behaviors enforced by the EVM and by Solidity. There are two noteworthy observations. First, even though all arithmetic operations performed by the EVM are modulo $2^{256}$, the result of $a + b$ in Figure 3.3 will silently wrap around if the value is larger than $2^{32} - 1$. This behavior is enforced by Solidity, not by the EVM. Second, division (or modulo) by zero results in 0. In other programming languages this would result in an exception being raised. However, in the EVM and Solidity versions prior to 0.4.0 this results in 0. Since most developers would expect an exception, starting from version 0.4.0 the Solidity compiler injects an invalid operation to throw an assert-style exception, causing the EVM to revert all changes.

```
1 function add(uint32 a, uint32 b) public returns(uint) {
2     return a + b;
3 }
```

**Figure 3.3:** An example of an integer overflow bug in Solidity.

### 3.2.2 Truncation Bugs

Converting a value of one integral type to a narrower integral type that has a shorter range of values may introduce so-called *truncation bugs*. Truncation bugs became infamous due to a 64-bit integer value that was converted to a 16-bit integer, which ultimately led to the explosion of an Ariane 5 rocket in 1996. While truncation bugs in smart contracts will (hopefully) not lead to explosions, they may lead nevertheless to a loss of precision, which ultimately may result in the loss of funds. For instance, consider the fallback function in Figure 3.4. The function converts and stores the received amount of ether to an unsigned integer of 32 bits. The value of `msg.value` is of type the `uint`, which is equivalent to the type `uint256`, thus it can hold integer values ranging from 0 to $2^{256} - 1$. If a caller transfers an amount larger than $2^{32} - 1$, this value will be truncated and the balance will be lower than the amount that the caller effectively transmitted.

```
1 mapping(address => uint32) balance;
2
3 function() public payable {
4   balance[msg.sender] = uint32(msg.value);
5 }
```

**Figure 3.4:** An example of a truncation bug in Solidity.

### 3.2.3 Signedness Bugs

Lastly, converting a signed integer type to an unsigned type of the same width (or vice versa) may introduce so-called *signedness bugs*. This conversion may change a negative value to a large positive value (or vice versa). For example, consider the function `withdrawOnce` in Figure 3.5. This function allows a caller to withdraw only once a maximum amount of 1 ether from the smart contract's current balance. However, if the parameter `amount` is a negative value, it will pass the bounds check, be converted to a large unsigned integer, and finally be passed as parameter to the `transfer` function. As a result, the `transfer` function will transfer an amount larger than 1 ether to the caller.

```
1 function withdrawOnce(int amount) public {
2   if (amount > 1 ether || transferred[msg.sender]) {
3     revert();
4   }
5   msg.sender.transfer(uint(amount));
6   transferred[msg.sender] = true;
7 }
```

**Figure 3.5:** An example of a signedness bug in Solidity.

## 3.3 Methodology

Due to the lack of publicly available source code, our goal is to detect integer bugs at the bytecode level. However, there are a number of challenges that we need to overcome in order to be able to detect integer bugs at the bytecode level. In this section, we describe our approach towards inferring integer types, detecting integer bugs, applying taint analysis to reduce false positives, and other challenges such as the identification of benign checks for integer bugs.

### 3.3.1 Type Inference

Type information such as integer size (e.g., 32 bits for `uint32`) and signedness (e.g., *signed* for `int`), is essential in order to check whether the result of an integer operation is in-bound or out-of-bound. However, type information is usually only available at the source code level and not at the bytecode level. That being said, due to certain code optimizations introduced by the Solidity compiler during compile time, it is actually possible to infer the size and the sign of integers at the bytecode level. For example, the compiler introduces for unsigned integers an `AND` bitmask in order to "mask off" bits that are not in-bounds with the integer's size. A zero masks the bit, whereas a one leaves the bit as it is. As an example, a `uint32` will result in an `AND` using `0xffffffff` as its bitmask. Thus, from the `AND` instruction we infer that it is an unsigned integer and from the bitmask we infer that its size is 32 bits. For

signed integers, the compiler introduces a sign extension via the SIGNEXTEND instruction. A sign extension is the operation of increasing the number of bits of a binary number while preserving the number's sign and value. In two's complement, this is achieved by appending ones to the most significant side of the number. The number of ones is computed using the following formula: $256 - 8(x + 1)$, where $x$ is the first value passed to the SIGNEXTEND instruction. As an example, consider an int32 which will result in a SIGNEXTEND instruction using the value 3 as its first parameter. Thus, from the SIGNEXTEND we learn that it is a signed integer and from the value 3 we infer that its size is 32 bits, by solving the following equation: $8(3 + 1)$.

### 3.3.2 Finding Integer Bugs

We now describe the techniques that we use to find the different types of integer bugs described in Section 3.2.

**Arithmetic Bugs.** For each arithmetic instruction that could potentially overflow (i.e., ADD and MUL) or underflow (i.e., SUB), we emit a constraint that is only satisfied if the in-bounds requirements are not fulfilled (see Table 3.1). As an example, if we have an addition of two unsigned integers $a$ and $b$, we emit a constraint to the solver that checks if $a + b > 2^n - 1$, where $n$ denotes the largest size of the two values, e.g., in case $a$ is a uint32 and $b$ is a uint64, $n$ will be $64$. Similarly, for signed/unsigned division (i.e., SDIV and DIV) and signed/unsigned modulo (i.e., SMOD, MOD, ADDMOD and MULMOD), we check whether the in-bounds requirements are not fulfilled as defined in Table 3.1. As an example, for signed division we emit a constraint that checks whether the divisor can be zero. If the solver can satisfy any of the emitted constraints under the current path conditions, we know that an arithmetic bug such as an overflow or a division by zero is possible.

**Truncation Bugs.** Solidity truncates signed and unsigned integers using SIGNEXTEND and AND instructions, respectively. For each instruction, we check whether it is possible for the input to be outside the range of the output. We do this by adding a constraint to the solver that is satisfied when the input value is larger than the output value. Moreover, we check the truncator value against two patterns, in order to detect and ignore truncations that have been intentionally introduced by Solidity. First, we check whether the binary representation of the truncator is equivalent to 160 ones. This represents a conversion to the type address. The second pattern consists in checking whether the binary representation of the truncator contains any zeros (ignoring leading zeros). This pattern aims at filtering out truncations that have been introduced by Solidity in order to squeeze multiple variables in one data storage slot.

**Signedness Bugs.** We reuse the approach by Molnar et al. [15], and adapt it to detect

signedness bugs in Ethereum smart contracts. The idea is to reconstruct signed/unsigned type information on all integral values, from the executed EVM instructions. This information is present in the source code but not in the bytecode. The algorithm to infer this information automatically, works as follows: consider four different types for integer values: “Top”, “Signed”, “Unsigned”, “Bottom”. “Top” means the value has not been observed in the context of a signed or unsigned integer; “Signed” means that the value has been used as a signed integer; “Unsigned” means the value has been used as an unsigned integer; and “Bottom” means that the value has been used inconsistently as both. These types form a lattice of four points. Our goal is to find symbolic variables that have the type “Bottom”. Every variable starts with the type “Top”. During execution we modify the type of a variable, based on the type constraints of certain instructions. For example, a signed comparison (e.g., `SLT`, `SGT`, etc.) between two variables causes both variables to receive the type "Signed", whereas an unsigned comparison (e.g., `LT`, `GT`, etc.) between two variables causes both variables to receive the type "Unsigned". Any variable that received both a signed and unsigned type, receives the type “Bottom”.

### 3.3.3 Taint Analysis

Taint analysis is a technique that consists in tracking the propagation of data across the control flow of a program. Taint analysis is extensively being used by numerous integer error detection tools in order to reduce the number of false positives [16, 14, 21, 26]. It is certainly possible to detect integer bugs without taint analysis. However, there are cases where integer bugs might be benign. For example, the Solidity compiler injects during compilation integer overflows at certain locations in the bytecode in order to optimize it for later execution. These overflows are intentional and should not be flagged as malicious. Taint analysis can help to distinguish between benign overflows introduced by the developer or compiler, and malicious overflows that are exploitable by an attacker. In taint analysis we have the notion of so-called *sources* and *sinks*, with the idea that data originates from a source and eventually flows into a sink. Taint is introduced by sources, which is subsequently propagated across the state of a program. In the case of the EVM, the program state consists of the stack, memory and storage. We follow a very precise approach on how taint should be propagated across stack, memory and storage, by taking the exact semantics of every EVM instruction into account (see Section 3.4.2). Sources are locations in a program, where data is originating from an untrusted input that might be controllable by an attacker, for example, environmental information or function parameters. Sinks represent locations, where data is used in a sensitive context, for example, security checks or access to storage. Thus, the attack surface of a smart contract is defined by the EVM instructions that are exposed to an attacker. In other words, an attacker is limited to certain sources in order to trigger bugs that are used in sensitive sinks. Therefore, by deliberately ignoring integer bugs that do not origi-

nate from a source and do not flow into a sink, we can focus exclusively on actual exploitable integer bugs and gracefully reduce the number of false positives, Figure 3.6 illustrates this process. We only check for integer bugs where the input data to the integer operation is tainted. Finally, we only validate an integer bug if it flows into a sink.

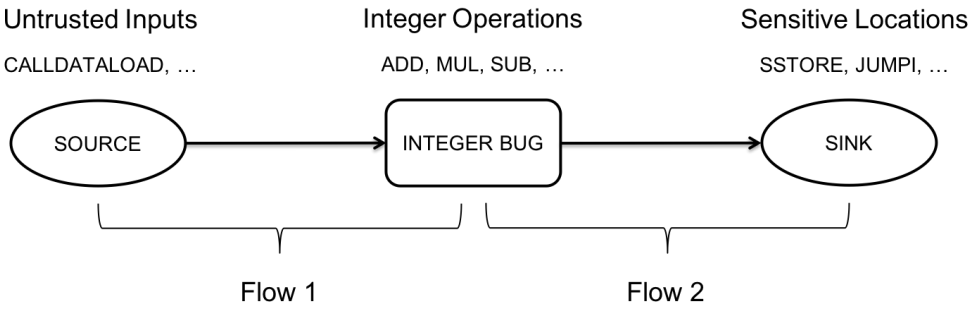



**Figure 3.6:** An integer bug is reported as valid if it originates from a source (i.e., untrusted input) and flows into a sink (i.e., sensitive location).

**Sources.** There are a number of EVM instructions, which an attacker could potentially use in order to introduce data that might lead to the exploitation of integer bugs. These instructions can be divided in: 1) block information, such as `GASLIMIT` or `TIMESTAMP`), 2) environmental information, such as `CALLER` or `CALLDATALOAD` and 3) stack, memory, storage and flow operations, such as `SLOAD` or `MLOAD`. However, many of these instructions have certain requirements and limitations which makes them almost impossible to be used by attackers in practice. For example, block information such as the `TIMESTAMP` can only be introduced by a miner and the proposed value can only have a divergence of 15 seconds from the timestamp of the other miners. Another example of a limited instruction is environmental information, such as the `CALLER`. An attacker can generate as many accounts as he wants, but he cannot predict the value of the account address. Thus, generating a desired address is essentially the same as brute-forcing. Taking all this into account, we selected `CALLDATALOAD` and `CALLDATACOPY` as sources for our taint analysis. The reasons are twofold: first, an attacker can pick any arbitrary value (he is solely limited by the data type chosen by the developer) and second, values are introduced at the transaction level and are therefore not only limited to miners.

**Sinks.** Whether or not an integer bug is harmful depends on where and how the smart contract uses the affected integer value. Such sensitive locations may originate from 1) stack, memory, storage and flow operations such as `SSTORE` or `JUMPI` and 2) system operations such as `CREATE` or `CALL`. We selected `SSTORE`, `JUMPI`, `CALL` and `RETURN` as sinks for our taint analysis, as these opcodes have an impact on path execution, storage and the sending of ether.

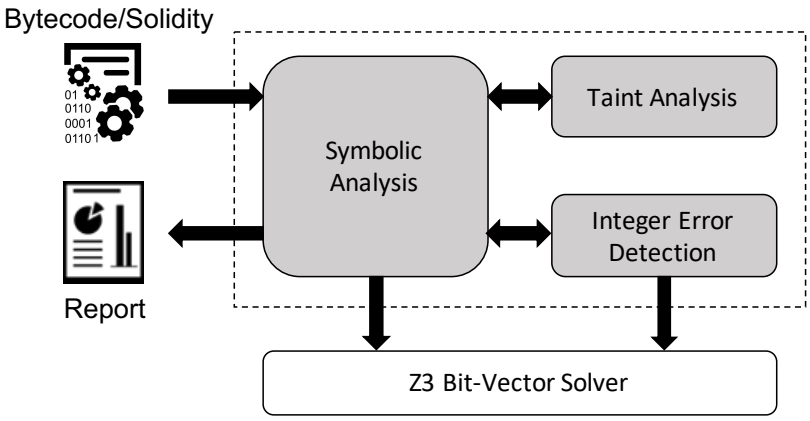



**Figure 3.7:** Architecture overview of OSIRIS. The shaded boxes represent its main components.

### 3.3.4 Identifying Benign Integer Bugs

Though taint analysis already reduces significantly the number of false positives, there are still some cases where an integer bug might originate from an untrusted source and flow into a sensitive sink, while being a benign integer bug. In order to cope with such cases, we came up with some heuristic rules that allow us to detect specific cases of benign integer bugs. For example, instead of immediately reporting an integer overflow or underflow as valid when we find it to be part of a branch condition, we check whether the predicate is designed to actually catch the bug. We note that common checks make use of the erroneous result to catch integer overflows and underflows, for example `if ((x + 1) < x)` or `if (x != (x * y) / y)`. We observe that these checks often use the same variable, on the right-hand side as well as on the left-hand side of the predicate. We also observe that if a predicate catches an integer bug, it is inclined to return soon or jump to a uniform error handling function. Hence, we report an integer bug as invalid if we find a predicate to use the same variable on the right-hand side as well as on the left-hand side, and one successor block of the branch condition in the control-flow graph ends in a `JUMPI`, `REVERT` or `ASSERTFAIL`.

## 3.4 OSIRIS

In this section, we provide an overview on the overall design and implementation details of OSIRIS[1].

### 3.4.1 Design Overview

Figure 3.7 depicts the architecture overview of OSIRIS. OSIRIS can take as input the bytecode or Solidity source code of a smart contract. The latter is internally compiled to EVM

[1]Code is publicly available at: https://github.com/christoftorres/Osiris

bytecode. OSIRIS outputs whether a contract contains any integer bug (e.g., overflow, underflow, truncation, etc.). OSIRIS consists of three main components: *symbolic analysis*, *taint analysis* and *integer error detection*. The symbolic analysis component constructs a Control-Flow Graph (CFG) and symbolically executes the different paths of the contract. The symbolic analysis component passes the result of every executed instruction to the taint analysis component as well as to the integer error detection component. The taint analysis component introduces, propagates and checks for taint across stack, memory and storage. The integer error detection component checks whether an integer bug is possible within the executed instruction.

### 3.4.2 Implementation

We implemented OSIRIS on top of OYENTE's [51] symbolic execution engine. OYENTE faithfully simulates 124 out of the 134 EVM bytecode instructions. The non-faithfully simulated instructions consist of logging operations (i.e., `LOG0`, `LOG1`, `LOG2`, `LOG3` and `LOG4`), operations regarding the output data from a previous contract call (i.e., `RETURNDATASIZE` and `RETURNDATACOPY`), the operation to create a new contract (i.e., `CREATE`) and operations to call other contracts (i.e., `DELEGATECALL` and `STATICCALL`). Non-faithfully simulated means that the engine faithfully simulates the stack, but does not implement the complete logic of the operation as described in [30]. However, since all of these operations (except the logging operations) are related to contract calls and detecting integer bugs across contract calls is out of scope, we can safely ignore the non-faithfully simulated instructions by OYENTE. OSIRIS is written in Python with roughly 1,200 lines of code (not counting OYENTE's symbolic execution engine). In the following, we briefly describe the implementation of each main component.

**Symbolic Analysis.** The symbolic analysis component starts by constructing a CFG from the bytecode, where nodes in the graph represent so-called basic blocks and edges represent jumps between individual basic blocks. A basic block is a sequence of instructions with no jumps going in or out of the middle of the block. OSIRIS can output a visual representation of the CFG depicting the individual path conditions and highlighting the basic blocks that include integer bugs (see Figure 3.8). After constructing the CFG, the symbolic execution engine starts by executing the entry node of the CFG. The engine consists of an interpreter loop that gets a basic block as input and symbolically executes every single instruction within that basic block. The loop continues until all the basic blocks of the CFG have been executed or a timeout is reached. In the case of a branch, the symbolic execution engine queries Z3 [11] in order to determine which path is feasible. If both paths are feasible, then the symbolic execution engine explores both paths in a Depth First Search (DFS) manner. Loops are terminated once they exceed a globally defined loop limit.

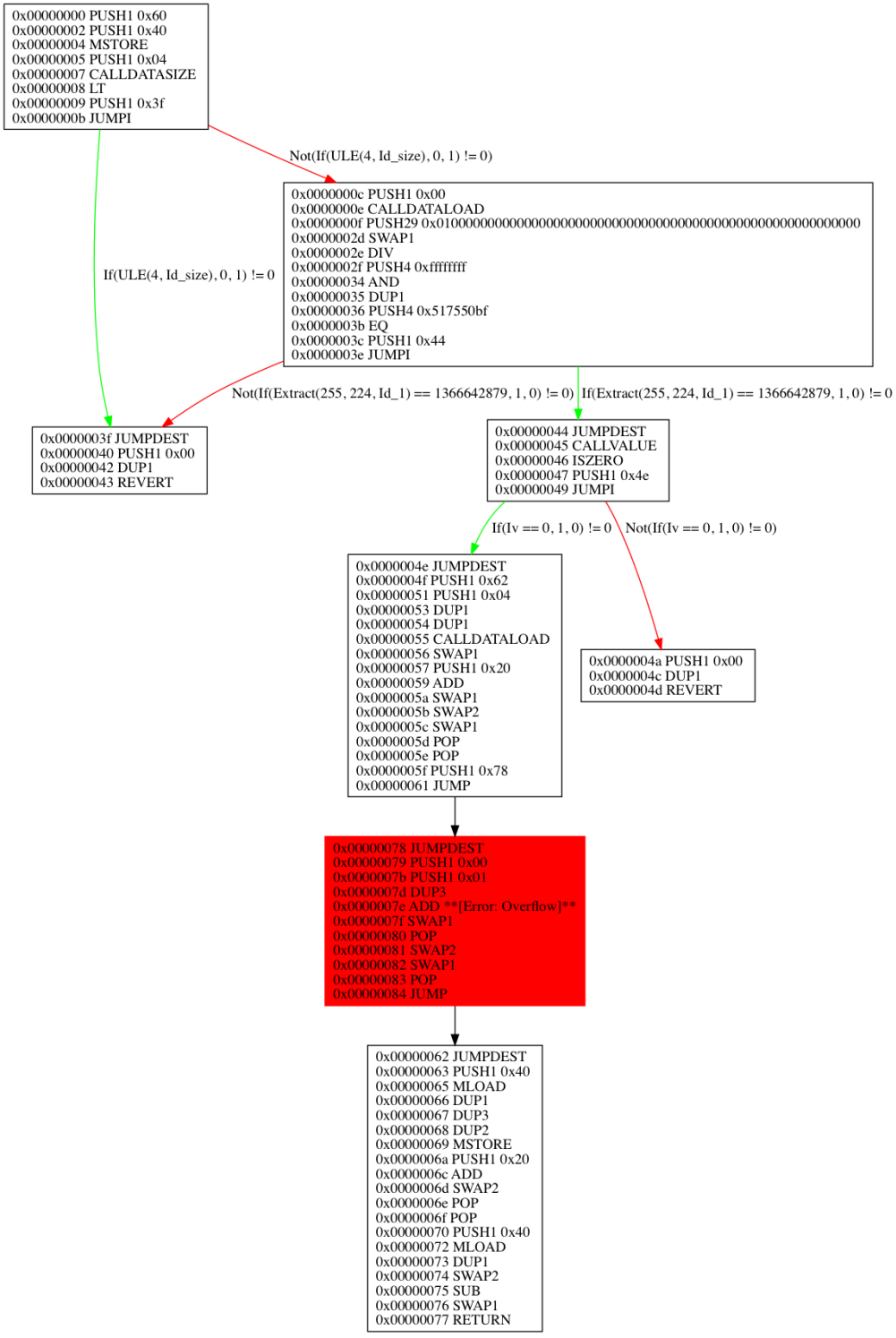



**Figure 3.8:** A representation of the control-flow graph that OSIRIS produces for Figure 3.9. The basic block highlighted in red indicates the location where an overflow may occur.

```
1 pragma solidity ^0.4.21;
2
3 contract Test {
4   function overflow(uint value) public pure returns(uint) {
5     return value + 1;
6   }
7 }
```

**Figure 3.9:** An example of a smart contract possibly producing an integer overflow at line 5.

**Taint Analysis.** The taint analysis component is responsible for *introducing*, *propagating* and *checking* of taint. The symbolic execution engine forwards every executed instruction to the taint analysis component. Afterwards, the taint analysis component checks whether the executed instruction is part of the list of defined sources. If that is the case, the taint analysis component introduces taint by tagging the affected stack, memory or storage location. We faithfully introduce and propagate taint across stack, memory and storage. We implemented the stack using an array structure following LIFO logic. To represent memory and storage, we simply used a Python dictionary that maps memory and storage addresses to values. Since the EVM is a stack-based and register-less virtual machine, the operands of an instruction are always passed via the stack. Our taint propagation method identifies the operands of each EVM bytecode instruction and propagates the taint according to the semantics of each instruction as defined in [30]. The taint propagation logic tags according to the following principle: if an instruction uses a tainted value to derive another value, then the derived value becomes tainted as well. By following this principle, we achieve a more precise taint propagation than, for instance, MYTHRIL [120]. MYTHRIL propagates taint across the stack, but for certain instructions it does not propagate taint across memory or storage. For example, the instruction `SHA3` computes the Keccak-256 hash over a memory region that is determined by two operands that are pushed onto the stack: *offset* and *size*. MYTHRIL simply checks if at least one of the two operands is tainted. If so, it taints the result that is pushed onto the stack. OSIRIS on the other hand, does not check the operands, but the memory region. OSIRIS only taints the result, if at least one of the values that is stored in the given memory region is tainted. As a final step, the taint analysis component verifies if a taint flow occurred, by checking whether the executed instruction is part of the list of defined sinks and if any of the values it used has been tainted by an integer bug.

**Integer Error Detection.** In contrast to the taint analysis component, the integer error detection component is not called upon every executed instruction. The integer error detection component is only called at instructions that may result in integer bugs, such as arithmetic instructions. For example, integer overflow checks are only performed if the symbolic analysis component executes an `ADD` or a `MUL` instruction, whereas width conversion checks are only performed if the symbolic analysis component executes an `AND` or a `SIGNEXTEND` instruction. Moreover, calls to the integer error detection component are only performed if

at least one of the operands of the executed instruction is tainted. If these criteria are met, then the symbolic execution engine eventually forwards the executed instruction along with the current path conditions to the integer error detection component. Afterwards, the component follows the different techniques as described in Section 3.3.2 in order to detect the specific integer bugs. For example, in the case of an `AND` instruction with tainted operands, the symbolic analysis component will call the integer overflow detection method of the integer error detection component. The integer overflow detection method first tries to infer the sign and the width of the two integer operands as described in Section 3.3.1 and then creates a formula with a constraint that is only feasible if an integer overflow is possible under the current path conditions. This formula is afterwards passed on to the Z3 solver, which checks for its feasibility. If the solver finds a solution to the formula, then the integer error detection component knows that an integer overflow is possible and returns an error back to the symbolic analysis component. After that, the symbolic analysis component calls the taint analysis component, which then taints the result of the `AND` instruction where its source represents the discovered integer bug.

## 3.5 Evaluation

In this section, we assess the correctness and effectiveness of OSIRIS via an empirical analysis and demonstrate its usefulness in detecting real-world vulnerabilities in Ethereum smart contracts. The empirical analysis is separated in a qualitative and a quantitative analysis. Via the qualitative analysis we aim to determine the reliability of our tool by comparing our results with ZEUS [112]. Via the quantitative analysis we intend to demonstrate the scalability of OSIRIS and to measure the overall prevalence of integer bugs contained in smart contracts that are currently deployed on the Ethereum blockchain.

**Experimental Setup.** All experiments were conducted on our high-performance computing cluster using 10 nodes with 960 GB of memory. Every node has 2 Intel Xeon L5640 CPUs with 12 cores each and clocked at 2,26 GHz, running a 64-bit Debian GNU/Linux 8.10 (jessie) with kernel version 3.16.0-4. We used version 4.6.0 of Z3, as our constraint solver for the symbolic execution engine as well as for our integer error detection module. For the symbolic execution engine we set a timeout of 100 ms per Z3 request. The global timeout for the symbolic execution was set to 30 minutes per contract. For our integer error detection module we set a timeout of 15 seconds per Z3 request. The loop limit, depth limit (for DFS), and gas limit for the symbolic execution engine was set to 10, 50, and 4 million, respectively.

### 3.5.1 Empirical Analysis

#### *Qualitative Analysis*

**Table 3.2:** Number of integer overflows and underflows detected by ZEUS and OSIRIS.

| Tool | Safe | Unsafe | No Result | Timeouts |
|---|---|---|---|---|
| ZEUS | 233 | 628 | 22 | 14 |
| OSIRIS | 711 | 172 | 0 | 35 |

**Dataset.** Kalra et al. [112] present a tool called ZEUS, which is capable of detecting integer overflows and underflows. The authors evaluate their tool using a dataset of 1,524 contracts that they obtained by periodically scraping explorers such as Etherscan, Etherchain and EtherCamp over a period of three months [111] on the main and test network. We decided to reuse this dataset in order to compare our results with ZEUS and to evaluate bugs that ZEUS does not detect such as division by zero or truncation bugs. However, the published dataset does not contain bytecode nor source code of the evaluated contracts. We were able to download the bytecode and source code for 961 out of the 1,524 contracts using Etherscan. Interestingly, 883 out of the 961 contracts are unique.

**Results.** We run OSIRIS on the 883 unique contracts and summarize our results for each of the three types of bugs below.

*Arithmetic Bugs.* We compare OSIRIS's capability of detecting integer overflows and underflows with ZEUS. Table 3.2 shows that OSIRIS reports most contracts to be safe whereas ZEUS reports most contracts to be unsafe. "Safe" means that no overflow or underflow has been detected, whereas "unsafe" means that either an overflow or an underflow has been detected. The reason for discrepancy between ZEUS and OSIRIS, is that OSIRIS aims at detecting solely overflows and underflows that are exploitable by an attacker in practice, thus limiting the number of reported bugs, while ZEUS aims to be complete. ZEUS reports no result for 22 contracts, where no result means either an error occurred or a timeout. ZEUS encountered less timeouts than OSIRIS, with 14 compared to 35. Table 3.3 depicts the confusion matrix of the evaluation between OSIRIS and ZEUS. OSIRIS reports 5 contracts to be unsafe, whereas ZEUS reports them to be safe. We manually verified these 5 contracts and indeed found them to potentially produce integer overflows. Figure 3.10 provides an example of a vulnerable function from one of the 5 contracts. The multiplication in the function

**Table 3.3:** Comparison between ZEUS and OSIRIS.

| | | OSIRIS | | |
|---|---|---|---|---|
| | | **Safe** | **Unsafe** | **No Result** |
| **ZEUS** | **Safe** | 228 | 5 | 0 |
| | **Unsafe** | 471 | 157 | 0 |
| | **No Result** | 12 | 10 | 0 |

`convertToWei` may overflow if `amount` is large enough. This questions ZEUS claim to be sound in terms of achieving zero false negatives. In 471 cases, ZEUS reports a contract to be unsafe while OSIRIS reports it to be safe. We manually analyzed all these cases and found that in some cases overflows were benign. These benign overflows were induced by the developer or by the Solidity compiler as part of handling data structures of dynamic size such as arrays, strings or bytes. The remaining overflows were indeed possible overflows, that were not caught by OSIRIS. OSIRIS could not catch them because they do not originate from the sources that we defined. Thus, technically OSIRIS could detect them by adding more sources, such as loading from storage (i.e., `SLOAD`). Apart from that, the authors of ZEUS state in their work that for several cases their tool reported unsafe, although the contract was safe. We encountered 32 of these cases. OSIRIS reports 28 of these cases to be safe, thus about 88% less than ZEUS. Unfortunately, ZEUS does not check for division by zero or modulo zero bugs, thus we cannot compare OSIRIS to ZEUS in this regard. OSIRIS did not find any modulo bugs. However, it did find 26 contracts vulnerable to division by zero bugs. We confirm the results via manual analysis of the source code and verifying that the bytecode was compiled using a compiler version lower than 0.4.0.

```
1 convertToWei(uint amount, string unit) external constant returns (uint) {
2   return amount * etherUnits[unit];
3 }
```

**Figure 3.10:** Overflow in *EtherUnitConverter*'s `convertToWei` function, not detected by ZEUS.

*Truncation Bugs.* OSIRIS reports 39 contracts carrying truncation bugs. We manually verified the findings and confirm the 39 bugs to be true positives. To confirm the findings, we checked the source code for type castings where integers are converted to smaller ranges.

*Signedness Bugs.* Signedness bugs seem to be less common. OSIRIS only reports 6 contracts to be vulnerable. Also here, we manually verified the findings and confirm the 6 bugs to be true positives. In order to confirm, we looked for conversions between signed and unsigned integers in the source code.

### *Quantitative Analysis*

**Dataset.** We collected the bytecode of 1,207,335 smart contracts by downloading the first 5,000,000 blocks from the public Ethereum blockchain. The timestamps of the collected smart contracts range from August 7, 2015 04:42:15 AM to January 30, 2018 1:41:33 PM. Figure 3.11 depicts the number of smart contracts in our dataset with respect to the month of their deployment on the blockchain. We state a sudden increase of smart contracts, starting from April 2017. In 2016, 50,980 contracts were deployed on average per month, whereas in

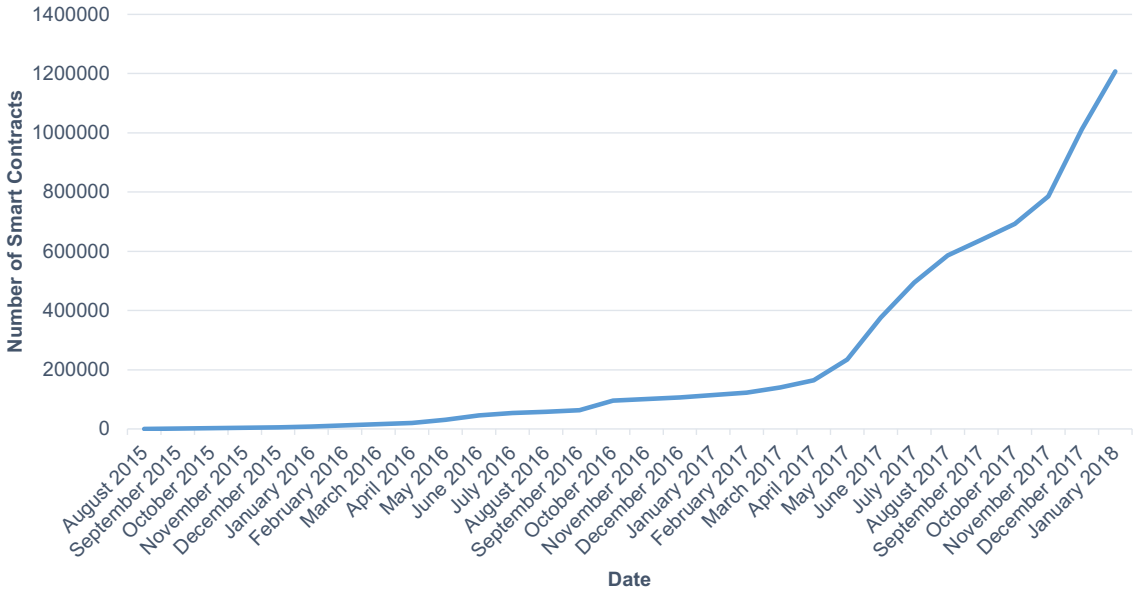



**Figure 3.11:** Number of smart contracts in Ethereum has increased abruptly.

2017 this number increased almost tenfold, with 447,306 contracts being deployed on average per month. Ethereum does not store the source code of smart contracts. To obtain the source code of a smart contract, users often refer to services such as Etherscan. However, at the time of writing, Etherscan solely lists the source code of 29,486 smart contracts [95]. Hence, only around 2% of the smart contracts on the Ethereum blockchain have their source code publicly available. Again, this emphasizes the need for tools such as OSIRIS, that are capable of analyzing smart contracts directly at the bytecode level. Out of these 1,207,335 contracts, only 50,535 are unique in terms of their bytecode. In other words, 96% of the smart contract on the Ethereum blockchain are copies.

**Performance.** On average, OSIRIS takes 75 seconds to analyze a contract, with a median of 13 seconds and a mode of 1 second. 524 contracts require more than 30 minutes to analyze. The number of paths explored by OSIRIS ranges from 1 to 1,394 with an average of 71 per contract and a median of 51. Similar to [51], we observe that the running time depends almost linearly on the number of explored paths. Finally, during our experiments, OSIRIS achieved a code coverage of about 88% on average.

**Results.** Figure 3.12 summarizes our results. OSRIS detects 42,108 contracts which contain at least one of the integer bugs discussed in Section 3.2. Out of these, 14,697 are distinct (by direct comparison of their bytecode). Figure 3.12 shows that most reported bugs are arithmetic (e.g., overflows, underflows, etc.) with 41,379 contracts as compared to 2,738 and 405 contracts for truncation and signedness, respectively. Out of these 41,379 contracts, 14,107 are found to be distinct, which account for roughly 28% of the 50,535

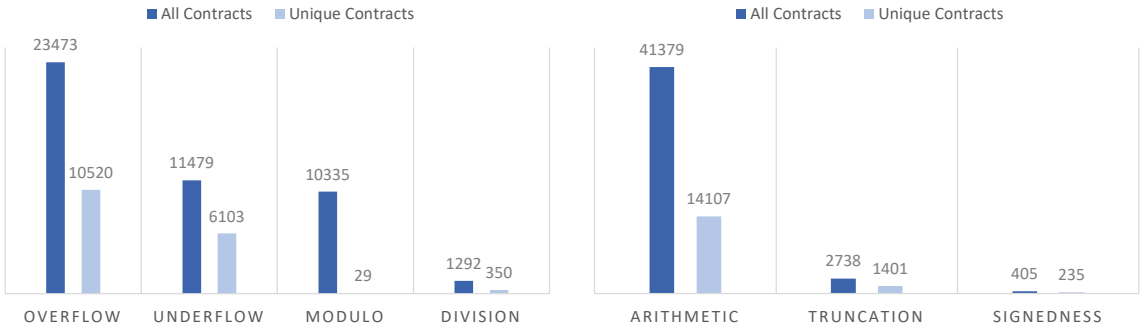



**Figure 3.12:** Number of vulnerable contracts reported by OSIRIS per integer bug type.

distinct contracts in our dataset. Figure 3.12 also depicts the distribution between reported arithmetic bugs. We note that overflows are the most common type of bugs with 23,473 vulnerable contracts, where 10,520 are distinct which account for about 21% of the distinct contracts in our dataset. Immediately after that follow underflows with 11,479 vulnerable contracts, where 6,103 are distinct which account for about 12% of the distinct contracts in our dataset. It is interesting to note that even though we only detect 29 distinct contracts vulnerable to modulo zero, the number of overall vulnerable contracts is 10,335. This implies that certain contracts are copied excessively and that one bug in such a contract can have a huge impact on the security of thousands of other contracts on the blockchain.

### 3.5.2 Detection of Real-World Vulnerabilities

In this section, we examine the effectiveness and usefulness of OSIRIS in detecting and reporting real-world vulnerabilities. For this purpose, we run OSIRIS on five divulged vulnerabilities and analyze 495 top Ethereum token smart contracts.

#### *Detecting Known Vulnerabilities*

**Table 3.4:** CVEs examined by OSIRIS.

| Token | Bug Name | CVE Number | Disclosed |
|---|---|---|---|
| BEC [94] | batchOverflow | CVE-2018-10299 | 22 April 2018 |
| SMT [99] | proxyOverflow | CVE-2018-10376 | 25 April 2018 |
| UET [101] | transferFlaw | CVE-2018-10468 | 28 April 2018 |
| SCA [100] | multiOverflow | CVE-2018-10706 | 10 May 2018 |
| HXG [97] | burnOverflow | CVE-2018-11239 | 18 May 2018 |

A security company called PeckShield [217] disclosed in 2018 five different vulnerabilities related to ERC-20 token smart contracts, each exploiting an integer overflow (see Table 3.4). OSIRIS successfully detects *all* the vulnerabilities listed in Table 3.4. From this small-scale

```
1 function transfer(address _to, uint256 _value) returns (bool success) {
2   if (balances[msg.sender] >= _value && _value > 0) {
3     balances[msg.sender] -= _value;
4     balances[_to] += _value;
5     Transfer(msg.sender, _to, _value);
6     return true;
7   } else { return false; }
8 }
```

**Figure 3.13:** Overflow at Line 4 in *StandardToken*'s `transfer` function.

experiment, we gain confidence that OSIRIS is suitable as a detection tool for vulnerabilities in real-world smart contracts.

#### *Detecting Unknown Vulnerabilities*

In the previous experiment, we analyzed OSIRIS's capability of effectively detecting known CVEs. In this experiment, we want to check whether OSIRIS is capable of detecting yet undiscovered vulnerabilities in Ethereum token smart contracts.

**Dataset.** Etherscan provides a list of top tokens ranked by their market capitalization [96]. As of June 2018, the list holds a total of 509 different tokens. Out of these, 495 have their source code publicly available. We downloaded the bytecode as well as the source code for these 495 smart contracts and analyzed them using OSIRIS.

**Results.** OSIRIS reported 164 contracts to be vulnerable, where 126 contracts were reported to contain overflows and 54 to contain underflows. We verified the findings via manual inspection of the source code. We found two overflows to be false positives and the rest of the findings to be indeed true positives. Although all of the reported overflows/underflows are semantically possible, yet most of them are unlikely to be exploited in practice. The reasons are twofold: 1) a large number of overflows and underflows may only be triggered by the owner of the smart contract and 2) a large number of overflows and underflows are due to implementations either not checking whether the balance of a receiver may overflow after a transfer (see line 4 in Figure 3.13), or not checking whether the value of the total supply may underflow before subtracting the amount of tokens to be burned (see line 4 in Figure 3.14). Nevertheless, two integer underflows reported by OSIRIS, have proven to be of particular interest. Let us consider the code snippet in Figure 3.15. The code originates from a token called *RemiCoin*[2]. OSIRIS reports that an integer underflow is possible at Line 11. The issue arises at the check at Line 7 (ironically commented as checking for allowance). The condition is not checking whether the amount is higher than the allowance, but whether the allowance is higher or equal to the amount. This is probably due to a simple copy-paste

[2] `https://etherscan.io/token/0x7dc4f41294697a7903c4027f6ac528c5d14cd7eb`

```
1 function burn(uint256 _value) returns (bool success) {
2   if (balances[msg.sender] < _value) return false;
3   balances[msg.sender] -= _value;
4   _totalSupply -= _value;
5   Burn(msg.sender, _value);
6   return true;
7 }
```

**Figure 3.14:** Underflow at Line 4 in function `burn`.

mistake, as some contracts have the exact same condition but return if the condition is false rather than when its true. Nevertheless, this subtle mistake has two tremendous consequences: 1) an attacker can transfer all the tokens from any address to another address of her own and 2) the attacker can provoke an underflow, hereby setting her allowance to any amount she desires. The same bug is also present in the UET token [101].

```
1  function transferFrom(address from, address to, uint value) returns (bool success) {
2    //checking account is freeze or not
3    if(frozenAccount[msg.sender]) return false;
4    //checking the from should have enough coins
5    if(balances[from] < value) return false;
6    //checking for allowance
7    if(allowed[from][msg.sender] >= value) return false;
8    //checking for overflows
9    if(balances[to] + value < balances[to]) return false;
10   balances[from] -= value;
11   allowed[from][msg.sender] -= value;
12   balances[to] += value;
13   // Notify anyone listening that this transfer took place
14   Transfer(from, to, value);
15   return true;
16 }
```

**Figure 3.15:** RemiCoin's `transferFrom` function allows an arbitrary user to steal tokens from another user.

RemiCoin (RMC) was released in 2017 and has a market capital of 27,520 USD. Its creators are unknown. At its peak in October 2017, RemiCoin was traded for 1.82 USD, whereas now its value has dropped to 0.0147 USD. At the time of writing, 348 addresses hold RemiCoins and a total of 11,497 transfers have been made so far. We checked whether this bug has been exploited in the wild. We found multiple transactions resulting in integer underflows[3]. However, we miss evidence of these being targeted attacks as the victims are still left with a rather high amount of tokens. Since the bug results in transactions with a legitimate allowance being refused, we find it quite surprising that this bug has not been

[3] `https://bit.ly/2LHeNf6`

noticed so far. Demonstrating the above attack on the public blockchain is feasible. However, for ethical reasons we were reluctant to do so. Therefore, we demonstrate the attack on a copy of the smart contract that we deployed on the Ropsten test network[4] and created two test accounts: 1) `0xe9131d546bba6e233b0a19e504179dc61365a77f` and 2) `0x7e2a886f1ba5942cc7a3a53fc6fae94868e318a0`. We deployed the contract via the first account, hence making this account the holder of the total supply of tokens. Afterwards, we performed our attack by calling the `transferFrom` function and passing as arguments the address of the first account, the address of the second account and finally the total supply of tokens[5]. As a result, the second account now owns all of the tokens and its allowance was set from zero to a substantial amount.

## 3.6 Discussion

In this section, we highlight weaknesses in the Ethereum ecosystem that lead to smart contracts that are prone to integer bugs. Further, we discuss possible remedies to prevent integer bugs from happening in smart contracts.

### 3.6.1 Causes for Integer Bugs

**Weaknesses of Solidity and EVM.** Solidity is a language that has been designed to lower the bar for developers entering the smart contract ecosystem. In that respect, its syntax resembles JavaScript, suggesting a dynamically typed scripting language, which in fact, it is not. Then again, during compilation from Solidity to EVM, the compiler warns about some type casts which gives the developer the impression of a strictly static type validation – which again is not true. In fact, Solidity compiles into static EVM bytecode, but the type system of Solidity does not strictly map into that of EVM. For example, although integers with less than 256 do not exist in EVM, Solidity attempts to give the developer the impression of different integer types by providing respective type identifiers and generating wrap-around behavior during compilation. This is a weakness because first, it suggests that developers could save memory by using shorter integer types and second, it makes the unexpected (integers wrapping around) the rule. As a second weakness we consider the overflow handling of EVM itself. Unlike in low-level programming, deliberate integer overflows is a rarely used feature in application development and we have not come across a single smart contract that uses integer overflow in a deliberate way. Nevertheless, neither the Solidity compiler nor EVM treat integer overflows as an exception but rather treat them as a real CPU would do – with some unexpected deviations such as feasibility of division by zero. Given the fact that aborting a smart contract will result in a safe rollback of the transaction, treating overflows

[4] `https://bit.ly/2HIKbrx`
[5] `https://bit.ly/2l7ITNy`

as an exception and panicking seems to be the safer alternative than silent wrapping.

**Unsafe Implementations of Standards.** The ERC-20 [32] token standard provides a standardized Application Programming Interface (API) for tokens within Ethereum smart contracts. The API provides basic functionality in order to transfer tokens, as well as to allow tokens to be approved such that they may be spent by another on-chain third-party. The standard describes an interface consisting of a number of functions and events, which a smart contract must implement in order to be compliant. The main issue with the current standard, is that it solely provides an interface. Its implementation is left to the developer of the token. As a consequence, many different implementations exist. Some implementations might have bugs and might be copied by other developers, with the bugs left unnoticed, hereby spreading the bugs across multiple contracts. In addition, some tokens introduce new functionality that is not part of the standard and hereby potentially introduce new bugs.

**Negligible and Incorrect Use of Safe Libraries.** In Section 3.5.2, we analyzed the safety of 495 token smart contracts. Token contracts perform a number of arithmetic operations such as subtracting from balances and adding to balances. However, these operations may produce integer bugs such as overflows and underflows. Therefore, it is recommended to perform such operations using a safe arithmetic library such as *SafeMath* [124]. SafeMath provides safe arithmetic operations for multiplication, division, addition and subtraction. We found that 337 out of the 495 contracts include the SafeMath library in their source code. Thus roughly 32% of the tokens do not make use of a SafeMath library and are therefore highly susceptible to overflows and underflows. Moreover, OSIRIS found 53 out of the 337 contracts to include bugs related to overflows and underflows. After manual inspection, we found that even when developers make use of the SafeMath library, this does not necessarily mean that they use it for every single arithmetic operation performed by their smart contract.

### 3.6.2 Ways Towards Safe Integer Handling

There are various ways to reduce the likelihood of potentially catastrophic integer bugs in Ethereum smart contracts. We discuss two different ways in the following:

(1) *Handle integer bugs at the application layer.* This is the approach taken by libraries such as SafeMath. This is already a best practice and the only way to avoid overflows without modifying the Solidity compiler or EVM. However, it comes at the price of additional EVM instructions which increases gas costs. Obviously not all developers see the benefit of using additional libraries for solving apparently simple arithmetic tasks.

(2) *Handle integer bugs at the compiler level*. Compiler-generated overflow checks remove the burden from developers but still create additional overhead in terms of gas costs and runtime performance. Other languages such as Rust go a route that combines

rigorous static checking with fail-fast at development time and defensive programming at runtime. This approach could be retrofitted to the Solidity compiler without affecting the language or the EVM: as we have shown in this chapter, static integer overflow checking of real-world smart contracts is feasible and could be integrated into the compiler to identify potential overflow bugs at development time (as it is done by [121] for C code, for example). By additional annotations such as `//@allow_overflow`, developers could explicitly mark variables that should be treated in an unsafe way to allow deliberate overflows. The drawback is obviously that generated EVM bytecode may still contain unnecessary and costly runtime checks.

## 3.7 Related Work

In the past few years, several approaches have been proposed to tackle the challenge of fully formalizing reasoning about Ethereum smart contracts. Numerous attempts have been made in modeling the semantics of Ethereum smart contracts in state-of-the-art proof assistants [71, 129, 34, 70, 104, 85]. Bhargavan et al. propose to translate a subset of Solidity to F* for formal verification [34]. This is similar to the approach initially followed by the Solidity compiler of translating Solidity contracts into WhyML to generate formal proofs for the why3 framework [129]. A number of alternative translations of EVM bytecode to manual assisted proofs have been proposed, including proofs in Coq [71] and Isabelle/HOL [70, 85]. While these approaches enable formal machine-assisted proofs of various safety and security properties of smart contracts, none of them provide means for fully automated analysis.

As a result, a large number of automated tools have been proposed for ensuring correctness and safety of smart contracts [51, 87, 122, 136, 112, 120]. All of these tools are based on symbolically executing EVM bytecode. Luu et al. were the first to present a symbolic execution tool called OYENTE [51]. The tool is capable of automatically detecting vulnerability patterns such as *transaction-ordering dependence*, *timestamp dependence*, *mishandled exceptions* and *re-entrancy*. Nikolic et al. present MAIAN [122], a tool that builds up on OYENTE and employs inter-procedural symbolic analysis as well as concrete validation in order to find and validate vulnerabilities on trace properties, such as *greedy*, *prodigal*, and *suicidal*, in Ethereum smart contracts. Tsankov et al. present SECURIFY [136], a tool that first symbolically analyses a contract's dependency graph to extract semantic information and afterwards checks for violations of safety patterns. To enable extensibility, the tool permits new patterns to be specified via a designated domain-specific language. In any case, none of the aforementioned tools currently check for integer bugs in smart contracts.

Kalra et al. propose ZEUS [112], a framework for automated verification of smart contracts using abstract interpretation and symbolic model checking, accepting user-provided policies. ZEUS inserts policy predicates as `assert` statements in the source code, then translates everything to an intermediate LLVM representation, and finally invokes its verifier

to determine assertion violations. The tool is capable of detecting integer overflows and underflows similar to OSIRIS, with the difference of OSIRIS working at the bytecode level and ZEUS at the source code level. However, source code is not always available. Moreover, ZEUS requires users to write policies to assert the security of smart contracts, which is sometimes not that trivial. Mueller et al. present MYTHRIL [120], a security analysis tool for Ethereum smart contracts. It uses concolic analysis, taint analysis and control flow checking to detect a variety of security vulnerabilities. MYTHRIL comes very close to the approach behind OSIRIS, with one of the differences being that OSIRIS uses a more precise and complete taint propagation logic while allowing users to define their own sources and sinks. Another difference is that MYTHRIL treats every integer as a 256-bit integer and therefore does not detect an overflow if for example two 32-bit integers are being added, OSIRIS on the other hand tries to infer the width of every integer in order to precisely tell if an arithmetic operation can overflow or not. Finally, at the time of writing, MYTHRIL seems to have issues in distinguishing between benign and malignant overflows and underflows [119]. OSIRIS effectively distinguishes between benign and malignant integer bugs. Ultimately, both ZEUS and MYTHRIL, fail to check for truncation bugs and signedness bugs, whereas OSIRIS does check for these two types of integer bugs.

## 3.8 Conclusion

We presented the design and implementation of OSIRIS – a symbolic execution tool for detecting integer bugs in Ethereum smart contracts that leverages taint analysis to reduce false positives. Our comparison with ZEUS [112] shows that ZEUS is not sound and that OSIRIS reports less false positives than ZEUS. For example, OSIRIS found 5 contracts to be unsafe whereas as ZEUS reported them to be safe. Our evaluation on over 1.2 million Ethereum smart contracts indicates that about 4% of the deployed contracts might be vulnerable to at least one of the three types of integer bugs presented in this chapter. Moreover, OSIRIS discovered integer bugs in two of the top 495 Ethereum token smart contracts. Finally, we also identified causes for integer bugs and proposed modifications to the EVM and the Solidity compiler to make smart contracts safer against integer bugs.

# 4 | ConFuzzius

## *Data Dependency-Aware Hybrid Fuzzing for Smart Contracts*

*In this chapter, we propose a data dependency-aware hybrid fuzzer for smart contracts. Although a variety of tools for detecting bugs in smart contracts have been proposed, most of these tools rely on symbolic execution, which may yield false positives due to over-approximation. Recently, a number of fuzzers have been proposed to detect bugs in smart contracts. Unfortunately, these tend to be more effective in finding shallow bugs and less effective in finding bugs that lie deep in the execution, therefore achieving low code coverage and many false negatives. An alternative that has proven to achieve good results in traditional programs is hybrid fuzzing, a combination of symbolic execution and fuzzing. In this chapter, we study hybrid fuzzing on smart contracts and present* ConFuzzius*, the first hybrid fuzzer for smart contracts.* ConFuzzius *uses evolutionary fuzzing to exercise shallow parts of a smart contract and constraint solving to generate inputs that satisfy complex conditions that prevent evolutionary fuzzing from exploring deeper parts. Moreover,* ConFuzzius *leverages dynamic data dependency analysis to efficiently generate sequences of transactions that are more likely to result in contract states in which bugs may be hidden. We evaluate the effectiveness of* ConFuzzius *by comparing it with state-of-the-art symbolic execution tools and fuzzers for smart contracts. Our evaluation on a curated dataset of 128 contracts and a dataset of 21K real-world contracts shows that our hybrid approach detects more bugs than state-of-the-art tools (up to 23%) and that it outperforms existing tools in terms of code coverage (up to 69%). We also demonstrate that data dependency analysis can boost bug detection up to 18%.*

## 4.1 Introduction

The inception of immutable, blockchain-based smart contracts has shown how to enable multiple mistrusting parties to trade and interact without relying on a centralized, trusted third party. The immutability of a contract is crucial. If at least one of the engaging parties were allowed to modify a digital contract, the contract's trust would vanish. Unlike traditional legal contracts, smart contracts do not allow a dispute resolution with a neutral third party. Most

importantly, smart contracts cannot be nullified — parties cannot revoke any deployed smart contract, even if its code figures undeniable software bugs. Therefore, this very immutability comes at a price: smart contracts must be tested extensively before exposing them and their users to significant monetary value. In the past, simple vulnerabilities (e.g., missing access control [80]) as well as more subtle vulnerabilities (e.g., reentrancy [57]) have led to losses exceeding many tens of millions of USD.

We can verify the behavior of a smart contract using four different approaches. (i) *Unit testing*: requires manual effort to cover the different sections of the code, but it unveils only a limited number of bugs within the test cases. (ii) *Symbolic execution*: analyzes contract behavior abstractly but performs slowly on complex contracts (i.e., path explosion problem). (iii) *Static analysis*: does not execute code and over-approximates the contract behavior — it can capture the entire contract execution surface, but it exhibits false positives that must be manually inspected. Finally, (iv) *fuzzing*: tests a contract reasonably fast by automatically generating test cases, with a generally lower false positive rate than static analysis. Fuzzing, however, can suffer from low code coverage, especially when inputs are fuzzed at random and hence does not overcome simple input sanity verification.

When fuzzing smart contracts, we face the following three challenges: 1) *input generation*, 2) *stateful exploration*, and 3) *environmental dependencies*. When it comes to input generation, the input space can be significantly broad. However, the solution might be limited to a specific point. For example, if a condition requires an input value of type `uint256` to equate to $42$, then the probability of randomly generating $42$ as input is tremendously small. Moreover, smart contracts are stateful applications, i.e., the execution may depend on a state that is only achievable following a specific sequence of inputs. Finally, the runtime environment of smart contracts exposes additional inputs related to the underlying blockchain protocol, such as the current block timestamp or other contracts deployed on the blockchain. As a result, the execution flow of smart contracts may also depend on environmental information besides transactional information.

We solve these three challenges as follows. In tandem with the fuzzing procedure, we employ symbolic taint analysis to generate path constraints on tainted inputs. Once we detect that the fuzzer is not progressing, we activate a constraint solver to solve the constraint in question. We collect this solution within a mutation pool, from which the fuzzer can draw to move past the challenging contract condition. Existing hybrid fuzzing approaches, e.g., Driller [58], cease the fuzzer when they are stuck and switch to concolic execution to get past the complex condition. Then, they restart the fuzzer once passed the condition. Our approach keeps the fuzzer running and only uses constraint solving to generate inputs on the fly, which will eventually be picked by the fuzzer via the mutation pools. Moreover, we perform path termination analysis to purge irrelevant inputs from the mutation pools. To deal with the statefulness of smart contracts, we chose to take advantage of the selection and crossover operators of genetic algorithms. Genetic algorithms follow three main steps:

*selection*, *crossover*, and *mutation*. The selection operator's task is to choose two individuals from the population, which are afterwards combined by the crossover operator to create two new individuals. The challenge here is to generate meaningful combinations of inputs. We leverage data dependencies between individuals to guide our selection and crossover operators in selecting/combining two individuals only if they follow a *read-after-write* (RAW) data dependency across state variables. Finally, to solve the third and last challenge, we instrument the execution environment (i.e., the Ethereum Virtual Machine) to fuzz environmental information and model the input to a contract as a tuple consisting of transactional *and* environmental data. In summary, this chapter makes the following contributions:

**Contributions**

- We propose ConFuzzius, the first design and implementation of a hybrid fuzzer for smart contracts.
- We present a novel method to efficiently create meaningful sequences of inputs at runtime by leveraging dynamic data dependencies between state variables.
- We evaluate ConFuzzius on a set of 128 curated smart contracts as well as 21K real-world smart contracts, and demonstrate that our approach not only detects more vulnerabilities (up to 23%), but also achieves more code coverage (up to 69%) than existing symbolic execution tools and fuzzers.

## 4.2 Methodology

This section discusses the three main challenges of fuzzing smart contracts via a motivating example and presents our solution towards solving these three challenges.

### 4.2.1 Motivating Example

Suppose a user participated in an Initial Coin Offering (ICO) on the blockchain and now owns a number of tokens. Now let us assume the user wants to sell a certain amount of their tokens at a variable price that increases 1 ether per day. Figure 4.1 shows a possible implementation of an Ethereum smart contract using Solidity. The smart contract allows a user to sell its tokens to an arbitrary user on the Ethereum blockchain. The contract sells the tokens to the first buyer willing to pay 42 ether, plus 1 ether for each day since the start of the sale. Moreover, the token sale should last no longer than 30 days. In this example, the smart contract acts as a simple mediator that automatically settles the trade between the user owning the tokens and the user willing to buy the tokens without both users requiring to know or trust each other. Smart contract based ICOs often follow a standard that is known as ERC-20 [32]. This standard provides an interface that standardizes function

```
1  interface Token {
2    function transferFrom(address sender, address recipient, uint256 amount) external
         returns (bool);
3    function allowance(address owner, address spender) external view returns (uint256);
4  }
5
6  contract TokenSale {
7    uint256 start = now;
8    uint256 end = now + 30 days;
9    address wallet = 0xcafebabe...;
10   Token token = Token(0x12345678...);
11   address owner;
12   bool sold;
13
14   function Tokensale() public {
15     owner = msg.sender;
16   }
17
18   function buy() public payable {
19     require(now < end);
20     require(msg.value == 42 ether + (now - start) / 60 / 60 / 24 * 1 ether);
21     require(token.transferFrom(this, msg.sender, token.allowance(wallet, this)));
22     sold = true;
23   }
24
25   function withdraw() public {
26     require(msg.sender == owner);
27     require(now >= end);
28     require(sold);
29     owner.transfer(address(this).balance);
30   }
31 }
```

**Figure 4.1:** Example of a vulnerable token sale smart contract. Lines highlighted in red represent complex conditions, whereas lines highlighted in gray illustrate read-after-write data dependencies and finally, lines highlighted in blue depict environmental dependencies.

names, parameters, and return values. For example, the standard includes a function called `transferFrom`, which allows a user to transfer a limited amount of tokens to an arbitrary user on behalf of the owning user. Another example is the function `allowance`, which returns the number of tokens that a user can spend on behalf of the owning user. The smart contract in Figure 4.1 works as follows. An arbitrary user can call the function `buy` to purchase the tokens for 42 ether and a fee of 1 ether for each day that has passed since the launch of the token sale. The contract will automatically transfer the tokens by calling the function `transferFrom` on the ICO's contract. After the purchase, the smart contract owner can call the function `withdraw` to retrieve the 42 ether and the fee of the purchase.

The contract contains two vulnerabilities, one known as *block dependency* and another one known as *leaking ether*. The latter is enabled via a bug in the function `Tokensale` (see line 14 in Figure 4.1). Prior to Solidity version 0.4.22, the only way of defining a constructor was to create a function with the same name as the contract. The function `Tokensale` is

supposed to be the constructor of the contract `TokenSale`. However, due to a typo, the names do not match, and the compiler does not consider the function as the contract's constructor. As a result, the function `Tokensale` is considered a public function that any user on the blockchain can call. This type of programming mistake has led to multiple attacks in the past [60]. The first vulnerability, namely block dependency, occurs when the transfer of ether depends on block information, such as the timestamp (see line 27 in Figure 4.1). A malicious miner can alter the timestamp of blocks that it mines. Although miners cannot set the timestamp smaller than the previous one, nor can they set the timestamp too far ahead in the future, developers should still refrain from writing contracts where the transfer of ether depends on block information. The second vulnerability, namely leaking ether, occurs whenever a contract allows an arbitrary user to transfer ether, despite having never transferred ether to the contract before. The following sequence of transactions triggers both vulnerabilities:

- $t_0$: A non-malicious user calls the function `buy` with a value equals to 42 ether + fee;
- $t_1$: An attacker calls the function `Tokensale`;
- $t_2$: The same attacker calls the function `withdraw` after 30 days.

When running the above example using ILF [160] (a state-of-the-art smart contract fuzzer), it is not capable of finding the two vulnerabilities even after 1 hour. Inspecting the code coverage reveals that ILF achieves only 39%. For comparison, CONFUZZIUS achieves roughly 95% code coverage and correctly identifies the two vulnerabilities in less than 10 seconds.

### 4.2.2 Input Generation

Generating meaningful inputs is crucial for automated software testing. Fuzzers generate inputs in order to execute not-yet-executed code. This generation can be completely random (black-box fuzzers) or driven by runtime information (grey-box fuzzers). In both cases, the primary approach is to mutate previous inputs to generate new inputs to test. Thus, finding the right heuristics is of fundamental importance to efficiently explore the target input space and, eventually, find latent bugs in the code. However, real-world programs tend to contain conditions that are hard to trigger. These complex conditions need to be addressed by fuzzers in order to execute as much code as possible. Line 20 in Figure 4.1 provides an example of such a complex condition. Function `buy` requires the transaction value to be equal to $42$ ether along with a variable fee that depends on the number of days that have past since the launch of the token sale. Figure 4.2 illustrates the Control-Flow Graph (CFG) of the function `buy` along with its branching conditions. The complex condition is highlighted in red in the CFG. A fuzzer following a traditional random strategy will fail to get past this condition since it will generate the desired value only once every $2^{256}$ trials.

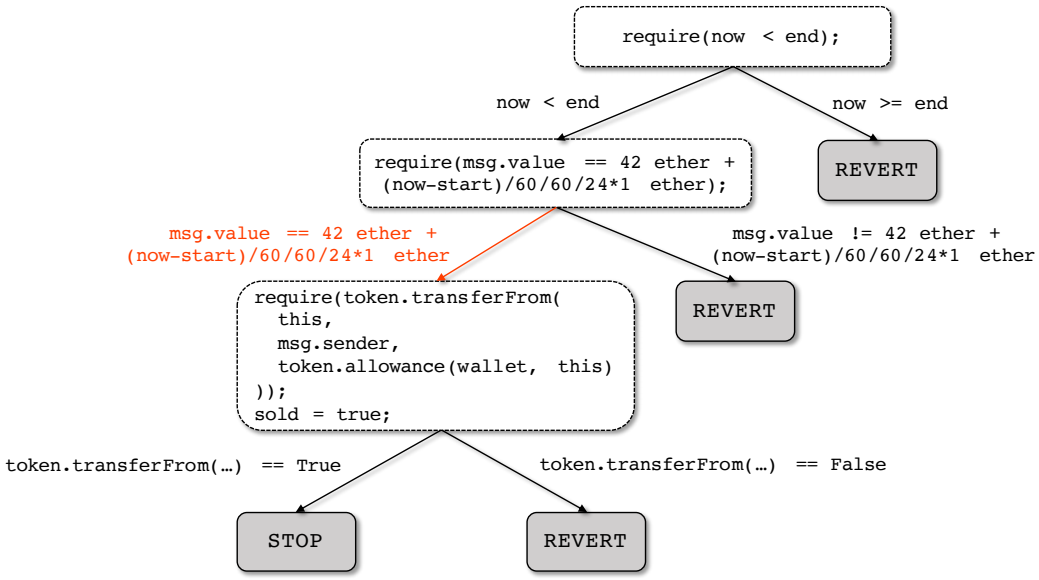



**Figure 4.2:** CFG of the function `buy()`. Complex path conditions are highlighted in red.

Existing smart contract fuzzers such as HARVEY [171] instrument the code and compute cost metrics for every branch to mutate the inputs. Our approach applies constraint solving to generate values for complex conditions on-demand. However, our fuzzer does not directly propagate these values, but instead stores them in so-called mutation pools. Mutation pools manage a set of values that the fuzzer can use to get past complex conditions. Every function has its own set of mutation pools, namely a mutation pool per function argument, transaction argument (e.g., transaction value), and environmental argument (e.g., block timestamp). Initially, all the pools are empty, and the fuzzer uses randomly generated values to feed the target functions. Once the fuzzer cannot discover new paths, it activates the constraint solver to generate new values. We use symbolic taint analysis to create the expressions required by the constraint solver to generate new values. We introduce taint in the form of a symbolic value whenever we come across an input during execution. This symbolic value is then propagated throughout the program execution, thereby forming step-by-step a symbolic expression that reflects the constraints on the particular input. Solving these expressions will result in new values that will be added to the mutation pools. The fuzzer will then pick these values from the mutation pools and generate new inputs that execute new paths. In the example provided in Figure 4.1, CONFUZZIUS will realize at a certain point that the code coverage is not increasing. It will then activate the constraint solver, which will output the value $42$ together with the current fee depending on the current block timestamp. The value will then be added to the mutation pool that manages the transaction value for the function `buy`. The value will be picked up from the mutation pool by CONFUZZIUS in the next fuzzing round, and the execution of the transaction will evaluate the condition at line 20 to `True`. This will result in CONFUZZIUS getting past the missing branch and executing new lines of code.

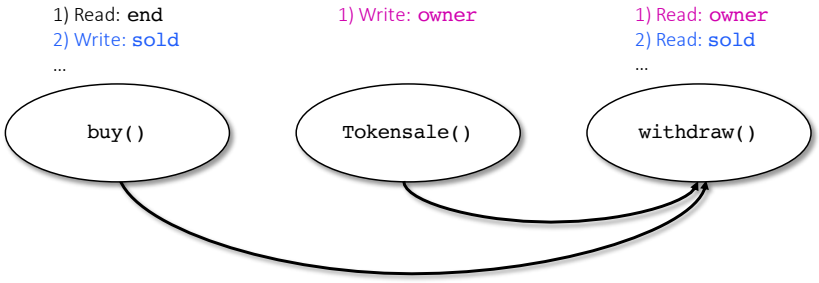



**Figure 4.3:** A dependency graph illustrating read-after-write (RAW) data dependencies contained in Figure 4.1. A node represents a smart contract function while an edge indicates a RAW dependency between two functions.

### 4.2.3 Stateful Exploration

Due to the transactional nature of blockchains, smart contract fuzzers must consider that each transaction may have a different output depending on the contract's current state, i.e., all the previously executed transactions. Combining multiple transactions together is necessary to generate states that trigger the execution of new branches. Ethereum smart contracts have, besides a volatile memory model, also a persistent memory model called *storage*, which allows them to keep state across transactions. For example, the global variables (i.e., state variables) `end`, `wallet`, `token`, `owner`, and `sold` in Figure 4.1 are storage variables and their values might change across transactions. Let us consider the two vulnerabilities mentioned earlier. An attacker will only be able to extract the funds via the function `withdraw`, if the variable `owner` contains the address of the attacker and the variable `sold` is set to `True`. However, this is only possible if the functions `buy` and `Tokensale` are called (i.e., executed) before the function `withdraw`. Thus only a particular combination of the three functions will trigger the two vulnerabilities. Although this example may seem straightforward, automatically finding the right combination of function calls within contracts with many functions can become challenging as the number of possible combinations grows exponentially. We base our solution on a simple observation: a transaction influences the output of a subsequent set of transactions if and only if it modifies a storage variable that one of the subsequent transactions will use. This property is a known data dependency called *read-after-write* (RAW) [19]. As a first step, ConFuzzius tracks during execution all the storage reads and writes performed by a transaction along with their storage locations. Afterwards, ConFuzzius combines transactions such that transaction $a$ is executed after transaction $b$ only if $a$ reads from the same storage location where $b$ writes to. ConFuzzius always executes the combination of transactions on a clean state of the contract. Thus, a transaction sequence contains only transactions that change the state used by one of the subsequent transactions within the same sequence by construction. In the example of Figure 4.1, ConFuzzius will progressively learn that:

- function `buy` reads storage variable `end` and writes to storage variable `sold`;
- function `Tokensale` writes to storage variable `owner`;
- function `withdraw` reads from storage variable `owner` and storage variable `sold`.

By leveraging the information learned above and combining transactions based on RAW dependencies, ConFuzzius will eventually create the following transaction sequence:

$$\texttt{buy()} \rightarrow \texttt{Tokensale()} \rightarrow \texttt{withdraw()}$$

The directed graph in Figure 4.3 depicts all RAW dependencies and possible combinations. The graph shows that the functions `buy` and `Tokensale` must be executed before the function `withdraw`, but that the order between the two can be arbitrary.

### 4.2.4 Environmental Dependencies

The execution of a smart contract does not only depend on the transaction arguments or the contract's current state. A smart contract's control-flow can also depend on input originating from the execution environment (e.g., a block's timestamp). Let us consider the contract in Figure 4.1. Even though the function `withdraw` has no input argument, the transfer of the balance is bound to some requirements. The requirement at line 27 is only satisfied if the transaction that triggered the function call is part of a block created 30 days after the contract's deployment. Thus, the condition is bound to the mining mechanism of the Ethereum blockchain. Miners are responsible for aggregating transactions from blockchain users into blocks and to broadcast them to other miners upon validation. When executing the transactions included in a block, the EVM accesses the block information contained therein. Block information includes the block hash, the miner's address, the block timestamp, the block number, the block difficulty, and the block gas limit. We solve this challenge by modeling this information as a fuzz-able input. These inputs follow the same fuzzing procedure as transaction inputs. We modified the EVM in order to be able to inject the fuzzed block information during the execution of the smart contract. However, modeling block information as fuzz-able inputs is not enough. The EVM also permits calls to other contracts deployed on the blockchain. Thus the control-flow of a smart contract may depend on the result of calling other contracts. Consider line 28 in Figure 4.1, where the state variable `sold` is required to be set to `True` in order for the attacker to be able to retrieve the funds. The variable `sold` can only be set to `True` if the two contract calls at line 21 (e.g., `token.allowance` and `token.transferFrom`) are successful. Similarly, we solve this challenge by instrumenting calls to contracts and modeling return values as fuzz-able inputs. Our modified EVM then injects the fuzzed return values at runtime.

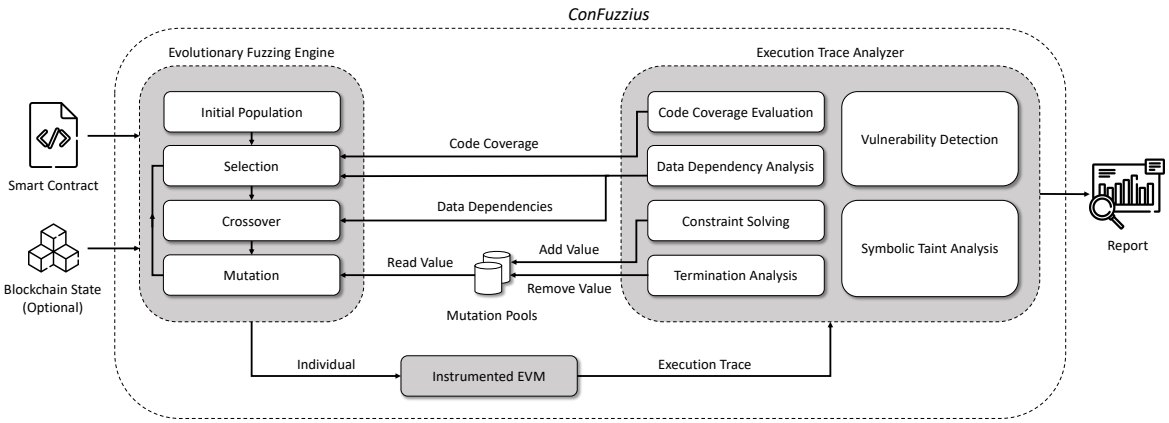



**Figure 4.4:** Overview of ConFuzzius's hybrid fuzzing architecture. The shadowed boxes represent the three main modules and form together a feedback loop.

## 4.3 ConFuzzius

In this section, we provide details on the overall design and implementation of ConFuzzius.

### 4.3.1 Architecture Overview

ConFuzzius's architecture consists of three main modules: the evolutionary fuzzing engine, the instrumented EVM, and the execution trace analyzer. Figure 4.4 provides a high-level overview of ConFuzzius's architecture and depicts its individual components. ConFuzzius has been implemented in Python with roughly 6,000 lines of code[1]. ConFuzzius takes as input the source code of a smart contract and a blockchain state. The latter consists of a list of transactions and is optional. The blockchain state is useful for fuzzing already deployed smart contracts or contracts that need to be initialized with a specific state. ConFuzzius begins by compiling the smart contract to obtain the Application Binary Interface (ABI) and the EVM runtime bytecode. Afterwards, the evolutionary fuzzing engine starts by generating individuals for the initial population, based on the smart contract's ABI. After that, the engine follows a standard genetic algorithm (i.e., selection, crossover, and mutation) and propagates the newly generated individuals to the instrumented EVM. The instrumented EVM executes these individuals and forwards the resulting execution traces to the execution trace analyzer. Next, the execution trace analyzer performs several analyses, e.g., symbolic taint analysis, data dependency analysis, etc. The execution trace analyzer is also responsible for triggering the constraint solver, running the vulnerability detectors, updating the mutation pools, and feeding information related to code coverage and data dependencies back to the evolutionary fuzzing engine. This process is repeated until at least one of the two termination conditions is met: a given number of populations has been generated, or a given amount of time has passed. Finally, ConFuzzius outputs a report containing information about the

[1]Source code is available at `https://github.com/christoftorres/ConFuzzius`.

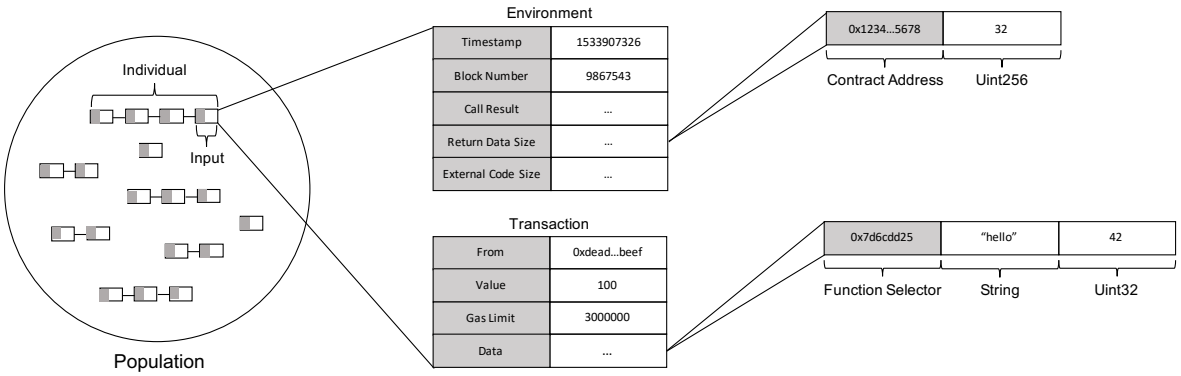



**Figure 4.5:** Encoding of our population and its individuals. The shadowed boxes depict immutable values, whereas the non-shadowed boxes depict mutable ones.

code coverage and the vulnerabilities that it detected.

### 4.3.2 Evolutionary Fuzzing Engine

In the following, we provide details on the encoding, initialization, fitness evaluation, selection, combination, and mutation of individuals.

**Encoding Individuals.** One of the most important decisions to make while implementing an evolutionary fuzzer is deciding on the representation of individuals. Improper encoding of individuals can lead to poor performance [3]. Figure 4.5 illustrates our encoding of individuals. Vulnerabilities are usually triggered either by sending a single transaction or a sequence of transactions to a smart contract. However, transactions alone are not enough to trigger all vulnerabilities (see Section 4.2.4). Specific vulnerabilities depend on the execution environment to be in a specific state. Thus, our encoding represents an individual as a sequence of inputs. Every input consists of an environment and a transaction. Both are encoded as key-value mappings. An environment includes block information such as the current timestamp and block number, but it also includes call return values, data sizes, and external code sizes. The latter three are encoded as an array of mappings, where a contract address maps to a mutable value (e.g., a call result or a size). A transaction includes the address of the sending account (*from*), the transaction amount (*value*), the maximum amount of gas for the contract to execute (*gas limit*), and the input data for the contract to execute (*data*). The input data is represented as an array of values where the first element is always the function selector, and the remaining elements represent the function arguments. The function selector is computed using the ABI and by extracting the first four bytes of the Keccak (SHA-3) hash of the function signature. As an example, the function `test(string a, uint b)`, has the string `test(string,uint)` as its function signature, which after hashing and extracting the first four bytes, results in `0x7d6cdd25` being its function selector.

**Initial Population.** The population is initialized with $N$ individuals, each of which initially contains only a single input (i.e., a single transaction and environment). The function selector to be included in the transaction is selected in a round-robin fashion. Function arguments are generated based on their type, which we obtain through the ABI. Depending on the type and size (i.e., fixed or non-fixed) of the argument, we apply different strategies to generate valid arguments for each function. For example, if the argument type is a fixed size `uint32`, then we randomly choose a value either from the valid input domain (e.g., between $0$ and $2^{32} - 1$) or from a set of inputs that trigger edge cases of the valid input domain (e.g., $0$, $1$, $4294967295$). The population is reinitialized whenever there has been no increase in code coverage for the past $k$ generations. This soft reboot introduces back diversity into the population and procrastinates premature convergence when the population has turned homogeneous.

**Fitness Evaluation.** The fitness evaluation of individuals plays a crucial role in evolutionary fuzzing. The computation of the fitness function is done repeatedly and must be therefore sufficiently fast. A slow computation can adversely make the fuzzing exceptionally slow. The fitness function is supposed to represent the landscape of the problem. In general, evolutionary fuzzers aim to achieve complete coverage of the code. While obtaining full code coverage does not necessarily mean that all vulnerabilities will be found, it is undoubtedly true that no vulnerabilities will be found in code that has not been explored. Our fitness function is based on branch coverage (a form of code coverage) and data dependencies. We define our fitness function for an individual $i$ as follows:

$$fit(i) = fit_{branch}(i) + fit_{RAW}(i) \tag{4.1}$$

The fitness $fit_{branch}$ is computed by counting the number of branches that remain unexplored by the individual. We keep track of all the branches that have been executed so far by all the individuals. Then, we iterate through the execution trace of the individual and analyze every conditional jump instruction (i.e., `JUMPI` instruction). A conditional jump always has two destinations, one for the `True` branch and one for the `False` branch. We obtain the jump destination of the `True` branch by extracting it from the stack and the jump destination of the `False` branch by incrementing the program counter by one. We increase the individual's fitness value $fit_{branch}$ by one for every jump destination that is not in our list of executed branches. This approach aims to prioritize individuals that require more exploration since these individuals will allow us to explore new parts of the contract. However, this metric alone is not enough. We are also interested in preserving individuals that allow us to create useful sequences of transactions (e.g., sequences with RAW dependencies), even though these individuals might have been already explored extensively. Therefore, we compute

the fitness $fit_{RAW}$, which takes this into account by using the data dependencies detected by our execution trace analyzer. We start with a $fit_{RAW}$ of zero and increment $fit_{RAW}$ by one for every write to storage that the individual has performed during execution. The final fitness value of an individual is then defined by the the sum of the two fitness values $fit_{branch}$ and $fit_{RAW}$. The combination of these two values allows us to drive the genetic algorithm to explore unexplored code while preserving individuals that are useful to explore deeper states of the smart contract.

**Selection.** The process of choosing two individuals for the crossover step is called selection. Literature proposes several selection operators [20]. We choose linear ranking selection as our selection operator since it considers the population as a whole during selection and not just a subset as it is, for example, the case in tournament selection. In linear ranking selection, individuals with a high fitness value are ranked higher than those with a low fitness value. In other words, an individual is selected with a probability that is linearly proportional to the individual's rank in the population. Hence, the worst individual has a rank of 1, the second-worst a rank of 2, the best-performing individual has a rank of $N$, where $N$ is the size of the population. All individuals have a chance of being selected, although the higher-ranked individuals will be slightly preferred. However, traditional linear ranking selection does not consider data dependencies between individuals. Therefore, we propose a modified version of the linear ranking selection strategy, where the first individual is selected based on linear ranking selection, however, the second individual is selected based on having a RAW dependency with the first individual following a round-robin fashion. In case there is no individual that has a RAW dependency with the first individual, we fallback to traditional linear ranking selection to select the second individual.

**Crossover.** The crossover operator creates two new individuals by recombining the input sequences of two existing individuals. Instead of randomly combining two individuals, we combine an individual after another only if the first performs a write to a storage location from which the second performs a read (RAW dependency). There are only two possible combinations in our case: individual $a$ appended to individual $b$, or vice versa, individual $b$ appended to individual $a$. If a combination yields a RAW dependency, then we combine both individuals by first selecting the individual whose input sequence performs the write and then append the individual whose input sequence performs the read. As opposed to traditional crossover, we are concatenating individuals rather than splitting them apart and swapping their input sequences. This preserves the RAW dependencies within the individuals themselves and creates individuals with new RAW dependencies. If there is no RAW dependency between two individuals, then we simply return one of the two individuals unmodified. However, it should be noted that individuals are not always combined even though they might have a RAW dependency. Individuals are combined based on a given crossover

probability $p_c$. Moreover, to prevent individuals from growing indefinitely large, we check before combining if the sum of their lengths exceeds the maximum size of $l$, and only combine them if their lengths are lower or equal to $l$. On the one hand, a small $l$ will produce shorter combinations of individuals, which will result in shorter execution times but also in finding less bugs. On the other hand, a larger $l$ will result in longer combinations of individuals, which will result in finding more bugs that lay deeper in the execution, but also in longer execution times. Therefore, a trade-off between completeness and performance must be taken when selecting an appropriate value for $l$.

**Mutation.** The mutation operator randomly modifies parts of a single individual in order to create a new individual. It introduces diversity in the population. Our mutation operator works by iterating through the sequence of inputs of an individual and mutating every environmental and transactional value based on a shared mutation probability $p_m$. A value can be mutated in two ways: replacing the original value with a random one or replacing the original value with a value from a *mutation pool*. Mutation pools act as a form of short-term memory. They allow the fuzzer to reuse values that have been previously observed or learned during past executions. There are in total nine different mutation pools, one per transactional and environmental value type. Hence, our fuzzer has a mutation pool for senders, amounts, gas limits, function arguments, timestamps, block numbers, call results, call data sizes, and external code sizes. All mutation pools are implemented to map a function selector to a circular buffer, except for the mutation pool on function arguments. The implementation is similar, except that we do not directly map the function selector to a circular buffer but to another mapping that maps to an argument index and then to a circular buffer. Thus, the pool for function arguments first maps to a function selector, then to an argument index, and then to a circular buffer. This is because functions can have more than just one argument, and we want to keep track of interesting values for every argument separately. Circular buffers help us ensure that the values contained therein are rotated in a round-robin fashion while old values are overwritten by newer ones (i.e., mimicking short-term memory). Our buffers can hold up to $10$ values by default. All mutation pools are initially empty, except for the mutation pool tracking transaction amounts, which is initialized with the values 0 and 1. When mutating a transactional or environmental value, we first check if the associated mutation pool is empty. If the pool is empty, we inject a randomly generated value based on the type of information extracted from the ABI. Otherwise, we inject the current value at the head of the circular buffer and rotate it.

### 4.3.3 Instrumented EVM

The EVM is responsible for executing the transactions generated by the individuals on the runtime bytecode of the contract that is under test. Its efficiency has a significant impact on the overall performance of the fuzzer. Hence, the EVM must achieve a high processing

rate of transactions. Every official Ethereum client implementation allows users to deploy smart contracts locally and send transactions to them. However, all of these clients require transactions to be encoded using the Recursive Length Prefix (RLP) format in order to be mined. We realized that the actual EVM execution time is negligible compared to the effort of encoding and decoding a transaction to and from the RLP format. Therefore, we decided to reuse an official Python implementation of the EVM [152] and incorporate it within our fuzzer. This removes the burden of mining blocks as well as encoding transactions and thus significantly speeds up the execution. Moreover, we slightly modified the EVM in order to be able to retrieve the execution trace of a transaction. An execution trace consists of an array, where every element contains the name of the executed instruction, the program counter, the execution stack, the call-stack depth, and a flag stating if an internal error has occurred during execution. The EVM itself is by default stateless and uses the blockchain to preserve states. However, since we are not interested in the internal persistence mechanism of the Ethereum blockchain, we decided to implement a simple storage emulator that is used by our EVM to persist the state changes that are performed during execution. All state changes are kept in memory to further improve the speed of execution. Besides persisting the state of smart contract executions, the storage emulator also allows us to inject custom environmental information such as the block timestamp or modify the result of a call. Finally, the storage emulator also enables us to create snapshots of the current state of the EVM. This allows us to quickly reset the state of the EVM to an initial state without having to redeploy the smart contract every time from scratch when executing the transactions of an individual.

### 4.3.4 Execution Trace Analyzer

The execution trace analyzer receives the execution trace from the instrumented EVM and then performs a number of analyses, such as code coverage evaluation, data dependency analysis, symbolic taint analysis, vulnerability detection, constraint solving, and termination analysis. Moreover, the execution trace analyzer also manages the values stored within the mutation pools and is responsible for providing code coverage information and data dependencies to the evolutionary fuzzing engine. Finally, it is also in charge of generating a report containing statistics about code coverage and vulnerabilities.

**Code Coverage Evaluation.** Code coverage is necessary for computing the fitness of an individual and detecting when the evolutionary fuzzing engine should activate constraint solving or reset the population. The code coverage is computed by counting the number of unique program counter values within the execution trace.

**Data Dependency Analysis.** Fitness evaluation, selection, and crossover require informa-

**Table 4.1:** Storage Layout of State Variables in Solidity.

| Variable Type | Declaration | Access | Storage Location |
|---|---|---|---|
| Primitive | `T v` | `v` | $s(v)$ |
| Struct | `struct v { T a }` | `v.a` | $s(v) + s(a)$ |
| Fixed Array | `T[10] v` | `v[n]` | $s(v) + n \cdot \lvert T \rvert$ |
| Dynamic Array | `T[] v` | `v[n]` | $h(s(v)) + n \cdot \lvert T \rvert$ |
| | | `v.length` | $s(v)$ |
| Mapping | `mapping(`$T_1$ `=>` $T_2$`) v` | `v[k]` | $h(k \mid\mid s(v))$ |

tion about data dependencies. The data dependency analysis tracks all the state variables, read from and written to, throughout the execution of the fuzzer. In contrast to existing approaches [173], which extract data dependencies via static analysis, our fuzzer retrieves access patterns to state variables at runtime (i.e., dynamically) by iterating through the execution trace and scouting for SLOAD and SSTORE instructions. The advantage of static analysis is that it is fast compared to dynamic analysis. However, the disadvantage is that it requires source code to precisely track data flows across variables. While data flows could be extracted from bytecode through static analysis, it would only work for simple variable types such as primitives and not for complex types such as mappings or arrays. Dynamic analysis on the other hand, allows us to track variables with complex types, even without source code. The disadvantage is the additional runtime overhead and implementational effort. The instruction SLOAD denotes a read from storage, whereas an SSTORE instruction denotes a write to storage. Table 4.1 depicts how Solidity computes the storage location for different types of state variables [190]. The function $s()$ determines the so-called *storage slot* of a particular variable $v$, whereas the function $h()$ computes a Keccak-256 hash. Statically-sized variables such as primitives, structs, and fixed-size arrays, are laid out contiguously in storage starting from position 0. Solidity uses a Keccak-256 hash computation to define the stored data's starting position due to the unpredictable size of dynamically-sized arrays and mappings. However, we are not interested in identifying individual storage locations but rather access to a particular variable. Therefore, our goal is to extract the storage slot $s(v)$ for a variable $v$, instead of the exact storage location. As an example, assume we have a state variable called `balances` of type mapping, which maps an `address` to a `uint`, and we have two addresses `a` and `b`. The storage location for `balances[a]` and `balances[b]` will be different, since the computation of the storage locations will be $h(a \mid\mid s(balances))$ and $h(b \mid\mid s(balances))$, respectively, where $\mid\mid$ means concatenation. However, these two storage locations share the same storage slot $s(balances)$, which enables us to link both storage locations together to the same state variable `balances`. Extracting the storage slot for statically-sized variables is straightforward as it is equivalent to the storage location. It can be achieved by merely popping the first element from the stack for both instructions, SLOAD and SSTORE. Extracting storage slots for mappings and dynamic arrays is more challenging.

As it is not possible to invert the result of a hash, we must keep track of the Keccak-256 hash computations by mapping the result of a SHA3 instruction to the memory contents that were involved in the computation. Note that the SHA3 instruction computes the Keccak-256 hash from a memory slice determined via two arguments on the stack, namely the memory offset and the memory size. Thus, for mappings, all we need to do is to obtain the mapped memory contents. To deal with concatenation, we only extract the last 32 bytes of the memory contents as these represent the storage slot. Finally, for dynamic arrays, we must keep track of the arithmetic addition of Keccak-256 hashes. The storage slot is then determined the same way as for mappings, except that there is no concatenation.

**Symbolic Taint Analysis.** The symbolic taint analysis produces symbolic constraints that are later used by other parts, such as constraint solving and vulnerability detection. We introduce taint in the form of symbolic values and track their flow across instructions. We leverage light dynamic taint analysis by injecting taint only for instructions that can be fuzzed, e.g., CALLDATALOAD, CALLVALUE, or TIMESTAMP. Taint is propagated across stack, memory, and storage. The propagation of taint across storage allows us to do inter-transactional taint analysis. We implemented the stack using an array structure that follows LIFO logic. We used a Python dictionary to map memory and storage addresses to values to represent memory and storage. Since the EVM is a stack-based, register-less virtual machine, the operands of instructions are always passed via the stack. Therefore, our taint propagation method identifies each EVM bytecode instruction's operands and propagates the taint according to each instruction's semantics as defined in the yellow paper [30]. The taint propagation logic follows an over-tainting policy, which tags the instruction's output as tainted if at least one of the instruction's inputs are tainted.

**Constraint Solving.** There are situations where the evolutionary fuzzing engine converges prematurely because it cannot advance past a complex conditional statement. The constraint solver's role is then to generate a valid input that allows the evolutionary fuzzing engine to get past the complex condition. The symbolic taint analysis tries to build a logical formula that describes the complex execution path, thereby reducing the problem of reasoning about the execution to the domain of logic. These logical formulas are often called path constraints. We implemented our own lightweight symbolic execution engine, that only executes instructions related to arithmetic operations (e.g., ADD, MOD, EXP), comparison logic (e.g., LT, EQ), and bitwise logic (e.g., AND, NOT). The engine consists of an interpreter loop that gets instructions from the execution trace and symbolically executes them. The loop continues until all the instructions contained in the execution trace have been executed. We obtain the formulas only for conditional statements with open branches, i.e., never executed branches. Each formula contains the path constraints to reach the conditional statement. We negate the last constraint, substitute the symbolic variables in the rest of the logical for-

mula with concrete values that have been used as inputs to trigger the execution trace, and use the Z3 SMT solver [11] to produce inputs to reach the open branch. Concretization helps us reduce the complexity of the formula and therefore avoid the path explosion problem. We then add the produced inputs to the mutation pools. Eventually, in one of the following generations, the mutation operator will pick up the solution, and our evolutionary fuzzing engine will now be able to get past the complex condition. We also keep the previously used inputs in the mutation pools, allowing the fuzzer to execute both branches.

**Termination Analysis.** The execution traces may contain valuable feedback on the validity of the inputs. Our fuzzer uses the execution traces to obtain feedback and learn whether an input is meaningful or not. The termination analysis inspects the execution traces for opcodes that indicate either correct or incorrect termination of execution. Invalid inputs will result in the execution trace terminating with a `REVERT`, `INVALID`, or `ASSERTFAIL` instruction, whereas valid inputs will result in the execution terminating with either a `SELFDESTRUCT`, `SUICIDE`, `STOP`, or `RETURN` instruction. Once we detect an incorrect termination, we analyze the last path constraint before the termination and retrieve the input values responsible for the incorrect termination. We then remove these values from the mutation pools. A transaction that results in an incorrect termination reverts all state changes made during execution and is therefore not relevant for creating meaningful RAW dependencies. This helps the fuzzer focus on more relevant parts of the code.

**Vulnerability Detection.** We detect vulnerabilities by analyzing the execution traces and the information returned by the symbolic taint analysis as well as the data dependency analysis. We define a detector per vulnerability. We implemented detectors for 10 different vulnerabilities. More detectors can be easily added to extend the detection capabilities of our fuzzer. We briefly elaborate on the implementation details of each detector:

- **Assertion Failure (AF).** We detect an assertion failure by checking if the execution trace contains an `ASSERTFAIL` or `INVALID` instruction.
- **Integer Overflow (IO).** Detecting integer overflows is not trivial, since not every overflow is considered harmful. Integer overflows may also be introduced by the compiler for optimization purposes. Therefore, we only consider an overflow as harmful, if it modifies the state of the smart contract, i.e., if the result of the computation is written to storage or is used to send funds. We follow our previous approach on detecting integer overflows [102] and start by analyzing if the execution trace contains an `ADD`, `MUL` or `SUB` instruction. We extract the operands from the stack and use these to compute the unbounded result of the arithmetic operation. Afterwards, we check if our result is equivalent to the result that has been pushed onto the stack. If they are not the same, we know that an integer overflow has occurred and we keep track of the overflow by

tainting the result of the computation. We report an integer overflow if the tainted result flows into an SSTORE instruction or a CALL instruction, as these instructions will result in updating the state of the smart contract.

- **Reentrancy (RE).** A reentrancy occurs whenever a contract calls another contract, and that contract calls back the original contract. We detect reentrancy by first checking if the execution trace contains a CALL instruction whose gas value is larger than 2,300 units and where the amount of funds to be transferred is either a symbolic value or a concrete value that is larger than zero. Finally, we report a reentrancy if we find an SLOAD instruction that occurs before the CALL instruction and an SSTORE instruction that occurs after the CALL instruction, where both instructions (i.e., SLOAD and SSTORE) share the same storage location.
- **Transaction Order Dependency (TD).** We detect transaction order dependency by checking if there are two execution traces with different senders, where the first execution trace writes to the same storage location from which the second execution trace reads.
- **Block Dependency (BD).** We detect a block dependency by checking if the execution trace contains either a CREATE, CALL, DELEGATECALL, or SELFDESTRUCT instruction, that is either control-flow or data dependent on a BLOCKHASH, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, or GASLIMIT instruction.
- **Unhandled Exception (UE).** We detect unhandled exceptions by first checking if the execution trace contains a CALL instruction that pushes to the stack the value 1 as a result of the call. A value of 1 means that an error occurred during the call (i.e., an exception). Afterwards, we check if the result of the call flows into a JUMPI instruction. If the result does not flow into a JUMPI instruction until the end of the execution trace, then we know that the exception of the call was not handled and we report an unhandled exception.
- **Unsafe Delegatecall (UD).** We detect an unsafe delegate call by checking if there is an execution trace that contains a DELEGATECALL instruction and terminates with a STOP instruction, but whose sender is an attacker address. Attacker and benign user addresses are generated at the start by the fuzzer.
- **Leaking Ether (LE).** We detect the leaking of ether by checking if the execution trace contains a CALL instruction, whose recipient is an attacker address and which has never sent any ether to the contract in a previous transaction or has never been passed as a parameter to a function by another address that is not an attacker.
- **Locking Ether (LO).** We detect the locking of ether by checking if a contract can actually receive ether but cannot send out any ether. To check if a contract cannot

send ether, we check if the runtime bytecode of the contract does not contain any CREATE, CALL, DELEGATECALL, or SELFDESTRUCT instruction. To check if a contact can receive ether, we check if the execution trace has a transaction value larger than 0 and terminates with a STOP instruction.

- **Unprotected Selfdestruct (US).** Similar to the leaking ether or unsafe delegatecall vulnerability detectors, this detector relies on attacker accounts. We detect an unprotected selfdestruct by checking if the execution trace contains a SELFDESTRUCT instruction where the sender of the transaction is an attacker and its address has not been previously passed as an argument by a benign user.

## 4.4 Evaluation

In this section, we evaluate the effectiveness and performance of ConFuzzius by answering the following three questions:

- Does ConFuzzius achieve higher code coverage than current state-of-the-art symbolic execution and fuzzing tools for smart contracts?
- Does ConFuzzius discover more vulnerabilities than current state-of-the-art symbolic execution and fuzzing tools for smart contracts?
- How relevant are ConFuzzius's individual components in terms of code coverage and vulnerability detection?

**Datasets.** We run our experiments using two different datasets. The purpose of the first dataset is to measure code coverage, whereas the second dataset aims to measure the detection of vulnerabilities[2]. The first dataset was obtained by crawling Etherscan's list of verified smart contracts [178]. These are real-world smart contracts where the source code is publicly available and verified to match the bytecode deployed on the Ethereum blockchain. We filtered out contracts that failed to compile using Solidity version 0.4.26, resulting in a dataset of 21,147 contracts. Moreover, we split our dataset into different clusters based on each contract's number of EVM bytecode instructions. The idea is to examine code coverage on smart contracts with different sizes. We used the standard k-means clustering algorithm to create the clusters. The number of clusters has been determined using the Elbow and the Silhouette method. Both methods yield 2 to be the optimal value for $k$. Table 4.2 lists the number of lines of source code (LoSC), number of public functions, and the number of EVM bytecode instructions for each cluster and the overall dataset. The first cluster represents small contracts ($\leq$ 3,632 instructions) and contains 17,803 contracts, whereas the second cluster represents large contracts ($>$ 3,632 instructions) and contains 3,344 contracts. The

[2]Code and datasets are publicly available at: https://github.com/christoftorres/ConFuzzius.

**Table 4.2:** Statistics of our real-world dataset and its two clusters.

| | | LoSC | | | Functions | | | Instructions | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Dataset** | **Contracts** | **Min** | **Max** | **Mean** | **Min** | **Max** | **Mean** | **Min** | **Max** | **Mean** |
| Small | 17,803 | 1 | 3,584 | 119 | 1 | 81 | 14 | 1 | 3,632 | 1,763 |
| Large | 3,344 | 44 | 3,148 | 429 | 1 | 190 | 32 | 3,633 | 16,889 | 5,495 |
| **Overall** | **21,147** | **1** | **3,584** | **168** | **1** | **190** | **17** | **1** | **16,889** | **2,353** |

second dataset is based on Durieux et al.'s curated dataset [177], a collection of annotated smart contracts, which the authors used to evaluate the effectiveness of smart contract analysis tools. Unfortunately, their curated dataset is missing certain types of vulnerabilities (e.g., assertion failures). We decided to reuse their dataset and extend it with annotated vulnerabilities from the Smart Contract Weakness Classification (SWC) registry [189]. We added contracts related to assertion failures, unsafe delegatecalls, leaking ether, locking ether, and unprotected selfdestructs. The extended dataset consists of 128 contracts with 148 annotated vulnerabilities.

**Baselines.** We compare ConFuzzius to the tools listed in Table 4.3. We limit our comparison to symbolic execution tools and fuzzers as we want to know if our hybrid approach performs better than these methods on their own. We chose Oyente [51] because of its popularity among the community and continuous development. We chose Mythril since a recent study on smart contract analysis tools [177] revealed that it performs better than a variety of existing tools (e.g., Manticore [87], SmartCheck [135], Securify [136], Maian [122], etc.). We chose M-Pro because it employs a similar transaction sequence combining strategy as ours, with the difference being that ours is dynamic, and theirs is static. We chose ILF [160] since it has proven to outperform existing smart contract fuzzers (i.e., ContractFuzzer [110] and Echidna [176]). We chose sFuzz because it is a recent work that has not been compared yet by previous works and because it is based on the popular fuzzing tool AFL. Table 4.3 compares the different types of vulnerabilities detected by each tool.

**Table 4.3:** Security tools evaluated in this work. Tools marked with ● support the detection of the vulnerability, while tools marked with ○ do not support the detection of the vulnerability.

| | | Requires | Requires | Vulnerability Detectors | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Toolname** | **Type** | **Source Code** | **ABI** | **AF** | **IO** | **RE** | **TD** | **BD** | **UE** | **UD** | **LE** | **LO** | **US** |
| Oyente [51] | Symbolic | ✗ | ✗ | ● | ● | ● | ● | ● | ○ | ○ | ○ | ○ | ● |
| Mythril[120] | Symbolic | ✗ | ✗ | ● | ● | ● | ● | ● | ● | ● | ● | ○ | ● |
| M-Pro[173] | Symbolic | ✓ | ✗ | ● | ● | ● | ● | ● | ● | ● | ● | ○ | ● |
| ILF [160] | Fuzzer | ✓ | ✓ | ○ | ○ | ○ | ○ | ● | ● | ● | ● | ● | ● |
| sFuzz[186] | Fuzzer | ✓ | ✓ | ○ | ● | ● | ○ | ● | ● | ● | ○ | ● | ○ |
| **ConFuzzius** | **Hybrid** | ✗ | ✓ | ● | ● | ● | ● | ● | ● | ● | ● | ● | ● |

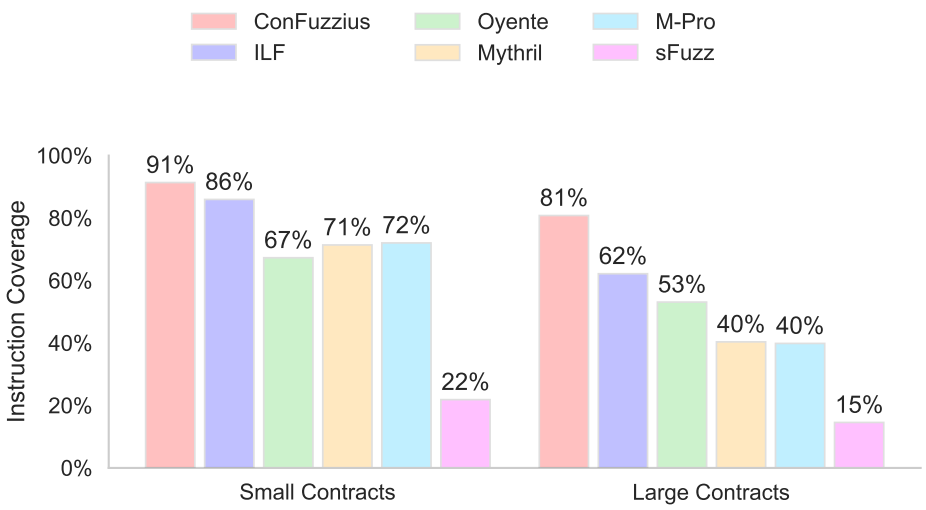



**Figure 4.6:** Overall instruction coverage of ConFuzzius and other tools.

We see that none of the tools are currently able to detect all of the ten vulnerabilities that are currently detectable by ConFuzzius. We also see that ConFuzzius does not requires source code as compared to the other fuzzers.

**Experimental Setup.** We followed the guidelines by Klees et al. [113] on evaluating fuzz testing. For each experiment, we performed 10 runs, each with independent seeds. For the first dataset we run experiments on the small contracts for 10 minutes each, whereas on the large contracts we run experiments for 1 hour each. Preliminary tests showed that most tools did not yield more coverage past these times. Moreover, before every run, we initialized ConFuzzius's and ILF's blockchain state with the same values that were used to deploy the contracts on the Ethereum mainnet. For the second dataset, we run the experiments for each contract for 10 minutes. We run our experiments on a cluster of 10 nodes, each with 128 GB of memory. Every node runs CentOS release 7.6.1810 and has 2 Intel® Gold 6132 CPUs with 14 cores, each clocked at 2.60 GHz. We run ConFuzzius with a variable population size that is computed as two times the number of functions contained in the ABI of the contract under test. We set the crossover probability and the probability of mutation to $0.9$ and $0.1$, respectively. The population is reinitialized whenever the code coverage does not increase for $k = 10$ generations. We set the maximum length for individuals to $l = 5$. Finally, we used Z3 version 4.8.5 as our constraint solver with a timeout of 100 milliseconds per Z3 request.

### 4.4.1 Code Coverage

Figure 4.6 depicts the overall instruction coverage (e.g., the average of all contract runs) of ConFuzzius and other security tools on the clusters of small and large contracts. Con-

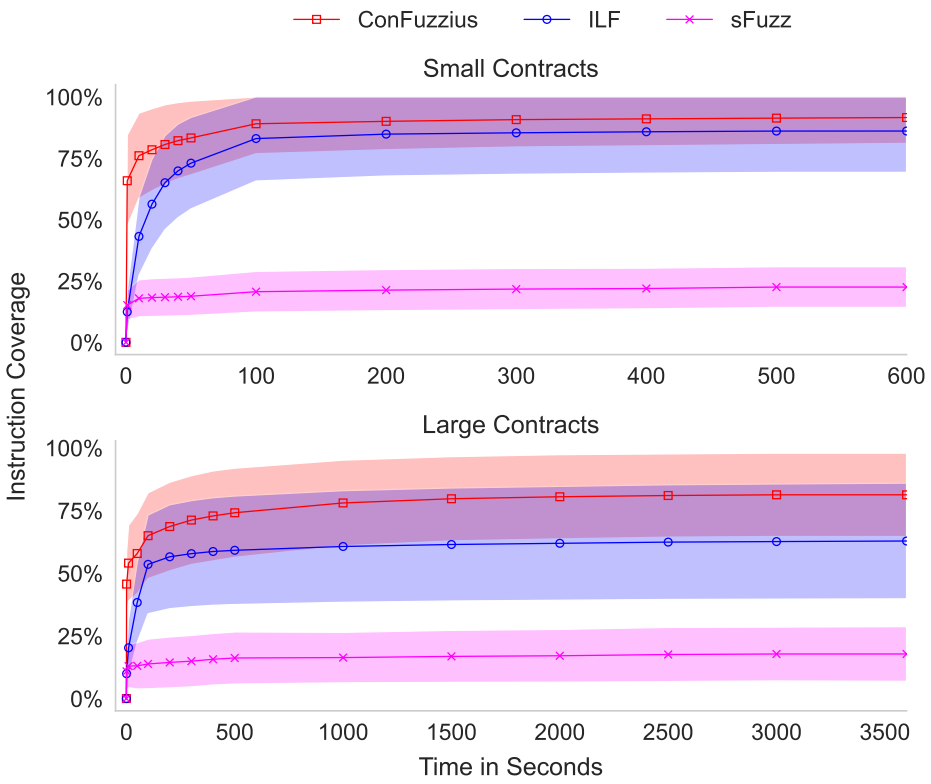



**Figure 4.7:** Overall instruction coverage of ConFuzzius, ILF and sFuzz over time.

Fuzzius achieves the highest coverage on the small and large contracts, with 91% and 81%, respectively. As expected, every tool struggles with the larger contracts. We see that the overall coverage is less for the larger contracts than for the smaller contracts. However, while the difference is roughly 10% for ConFuzzius, symbolic execution tools such as Mythril have a difference of 31%. Figure 4.7 compares the overall instruction coverage of ConFuzzius and the two other fuzzers, ILF and sFuzz, over time. We only plotted the instruction coverage for these three tools as we do not have coverage information over time for symbolic execution tools. ConFuzzius not only consistently outperforms ILF and sFuzz, but it also achieves more code coverage in a shorter time. On the small contracts, ConFuzzius achieves after 1 second 66% instruction coverage, whereas ILF and sFuzz achieve solely 12% and 15%, respectively. On the large contracts, ConFuzzius achieves after 1 second 46% instruction coverage, whereas ILF and sFuzz achieve only 10% and 11%, respectively.

### 4.4.2 Vulnerability Detection

Table 4.4 summarizes the vulnerabilities detected by each tool for each of the 10 categories on the extended curated dataset. Each entry shows the number of true positives (left-hand side) and false positives (right-hand side). For example, Oyente reported for assertion fail-

**Table 4.4:** True positives and false positives detected by each tool per vulnerability type.

| | Vulnerabilities | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Toolname** | **AF** | **IO** | **RE** | **TD** | **BD** | **UE** | **UD** | **LE** | **LO** | **US** | **Total** |
| OYENTE | 6/6 | 12/4 | 8/0 | 2/0 | 0/0 | - | - | - | - | 0/0 | 28 |
| MYTHRIL | 7/3 | 18/5 | 10/0 | 0/0 | 3/0 | 24/0 | 0/0 | 4/0 | - | 2/0 | 68 |
| M-PRO | 7/3 | 18/5 | 10/0 | 0/0 | 3/0 | 24/0 | 0/0 | 4/0 | - | 2/0 | 68 |
| ILF | - | - | - | - | 0/0 | 10/0 | 1/2 | 4/0 | 5/0 | 3/0 | 23 |
| SFUZZ | - | 12/0 | 7/0 | - | 1/0 | 21/0 | 1/2 | - | 0/0 | - | 42 |
| CONFUZZIUS | **10/0** | **18/0** | **10/0** | **2/0** | **7/0** | **46/0** | **1/0** | **4/0** | **5/0** | **3/0** | **106** |
| Total Unique | 14 | 19 | 11 | 4 | 7 | 75 | 1 | 9 | 5 | 3 | 148 |

ure (AF), 6 true positives and 6 false positives (i.e., 6/6), whereas MYTHRIL reported for assertion failure (AF), 7 true positives and 3 false positives (i.e., 7/3). Overall, we see that CONFUZZIUS detected the most number of vulnerabilities, namely 106 out of 148 vulnerabilities (roughly 71% of all vulnerabilities). ILF detected the least number of vulnerabilities, with 23 out of 148. In the following, we discuss the results obtained for each category.

**Assertion Failure (AF).** CONFUZZIUS detects more assertion failures than the other tools, namely 10 out of 14, and does not report any false positives. Both MYTHRIL and M-PRO report 7 assertion failures and 3 false positives. OYENTE reports 6 true positives and 6 false positives. Our manual investigation reveals that they over-approximate the satisfiability of execution paths due to incorrect modeling. For example, OYENTE reports in Figure 4.8 an assertion failure at line 8. However, this is not possible because `param` is set at the constructor and is checked to be always larger than zero.

```
1  contract AssertMultiTx1 {
2      uint256 private param;
3      constructor(uint256 _param) {
4          require(_param > 0);
5          param = _param;
6      }
7      function run() {
8          assert(param > 0);
9      }
10 }
```

**Figure 4.8:** False positive reported by OYENTE on an assertion failure.

**Integer Overflows (IO).** CONFUZZIUS reports the same number of integer overflows as MYTHRIL and M-PRO, namely 18 out of 19. However, MYTHRIL and M-PRO also report 5 false positives, whereas CONFUZZIUS reports none. For example, MYTHRIL reports in Figure 4.9 an integer overflow at line 6. However, there is no possibility to initialize `balanceOf`,

therefore an overflow can never occur because the `require` statement at line 4 will never be satisfied.

```
1 contract IntegerOverflowAdd {
2   mapping (address => uint256) balanceOf;
3   function transfer(address _to, uint256 _value) {
4     require(balanceOf[msg.sender] >= _value);
5     balanceOf[msg.sender] -= _value;
6     balanceOf[_to] += _value;
7   }
8 }
```

**Figure 4.9:** False positive reported by MYTHRIL on an integer overflow.

**Reentrancy (RE).** CONFUZZIUS, MYTHRIL and M-PRO detect the same number of reentrancy vulnerabilities, namely 10 out of 11. OYENTE and SFUZZ detect 8 and 7, respectively.

**Transaction Order Dependency (TD).** CONFUZZIUS and OYENTE detect 2 contracts vulnerable to transaction order dependency. Both MYTHRIL and M-PRO do not detect any of the 4 contracts to be vulnerable to transaction order dependency.

**Block Dependency (BD).** CONFUZZIUS is the only tool capable of detecting all 7 block dependencies. Our manual investigation reveals that CONFUZZIUS is capable of detecting more block dependencies because of its environmental modeling, which allows CONFUZZIUS to fuzz block information and therefore, CONFUZZIUS can detect more calls that are dependent on block information.

**Unhandled Exception (UE).** CONFUZZIUS reports the largest number of unhandled exceptions, namely 46 out of 75. MYTHRIL and M-PRO report 24 unhandled exceptions. ILF and SFUZZ report 10 and 21 unhandled exceptions, respectively. Similar to block dependency, CONFUZZIUS detects more unhandled exceptions because it models call return values as environmental information that can be fuzzed. This allows CONFUZZIUS to simulate exceptions and check if they are handled.

**Unsafe Delegatecall (UD).** CONFUZZIUS is the only tool that detects unsafe delegatecall without false positives. Neither MYTHRIL nor M-PRO were able to detect unsafe delegatecalls. ILF and SFUZZ detect an unsafe delegatecall, but also report 2 false positives. For example, in Figure 4.10 ILF reports an unsafe delegatecall at line 13. However, the variable `callee` can only be changed by the owner and the delegatecall is therefore safe.

**Leaking Ether (LE).** CONFUZZIUS, MYTHRIL, M-PRO and ILF detect the same number of ether leaking vulnerabilities, namely 4 out of 9. None of the tools report false positives.

```
1  contract Proxy {
2    address callee;
3    address owner;
4    constructor() {
5      callee = address(0x0);
6      owner = msg.sender;
7    }
8    function setCallee(address newCallee) {
9      require(msg.sender == owner);
10     callee = newCallee;
11   }
12   function forward(bytes _data) {
13     require(callee.delegatecall(_data));
14   }
15 }
```

**Figure 4.10:** False positive reported by ILF on an usafe delegatecall.

**Locking Ether (LO).** Both ConFuzzius and ILF detect all ether locking vulnerabilities. sFuzz, on the other hand, does not detect any of the vulnerabilities.

**Unprotected Selfdestruct (US).** Both ConFuzzius and ILF detect all the unprotected selfdestruct vulnerabilities. Oyente does not detect any of the vulnerabilities and Mythril as well as M-Pro, detect 2 of the 3 unprotected selfdestruct vulnerabilities.

### 4.4.3 Component Evaluation

In the following, we evaluate the importance of ConFuzzius's three main components: 1) *constraint solving*, 2) *read-after-write dependency analysis* and 3) *environmental instrumentation*. We randomly selected 100 contracts from each cluster. We then performed three experiments for each contract, where we deactivated a different component for each experiment. For example, in the "Without Constraint Solving" experiment, we disabled constraint solving but kept RAW dependency analysis and environmental instrumentation enabled. This means that inputs will be generated following a random uniform distribution, and the constraint solver will not be used to produce inputs. In the "Without Read-After-Write Dependency" experiment, we disabled RAW dependency analysis but kept constraint solving and environmental instrumentation enabled. This means that transaction sequences will not be combined following RAW dependencies, but that they will be combined at random, following a uniform distribution. Finally, in the "Without Environmental Instrumentation" experiment, we disabled environmental instrumentation but kept constraint solving and RAW dependency analysis enabled. This means that environmental information such as block timestamps or call return values will not be fuzzed. We repeated each experiment 10 times and run the experiments on the small contracts for 10 minutes, and on the large contracts for 1 hour. Figure 4.11 depicts our results. Each bar shows the percentage of the achieved results compared to the results when all three components were enabled (i.e., the grey bar

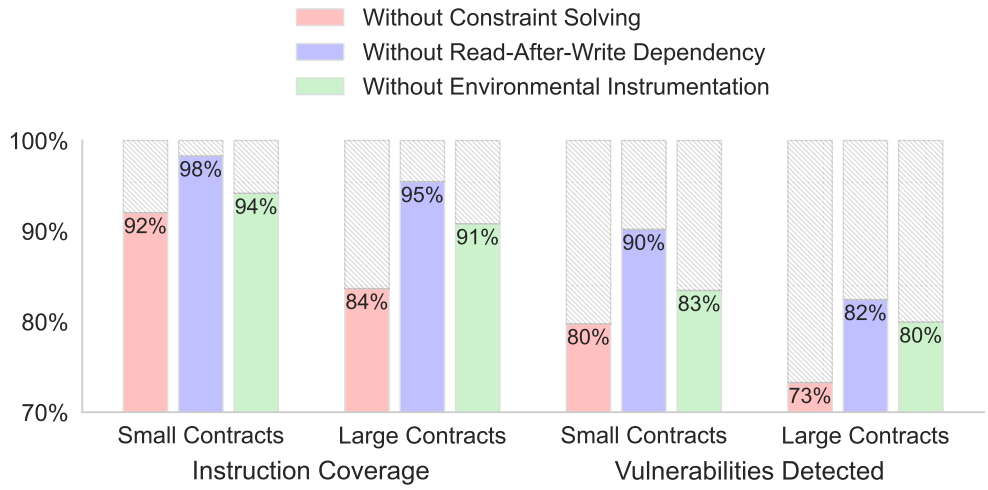



**Figure 4.11:** Comparison of overall instruction coverage and vulnerabilities detected between ConFuzzius's individual components.

in the back). We see that each component is an added value for ConFuzzius, which means that they are all essential to ConFuzzius's performance. However, we can see that generating meaningful inputs via constraint solving plays an essential part in achieving broad code coverage and detecting more bugs. Also, environmental instrumentation seems to help achieve code coverage and detect bugs. Nevertheless, our novel method that leverages RAW dependency analysis to create meaningful sequences of inputs at runtime can provide 2% more code coverage on small contracts and 5% on large contracts. Further, it allows the detectors to find 10% and 18% more vulnerabilities in small and large contracts, respectively.

## 4.5 Related Work

Since its introduction by Miller et al. [4], fuzzing has been applied to many different domains and targets. American Fuzzy Loop (AFL) [52] is one of the most widespread fuzzers and it is based on evolutionary fuzzing and exploits execution data to guide the generation/mutation of fuzzed inputs. Besides AFL and its offsprings [64, 63], other fuzzers also use evolutionary approaches to generate test inputs automatically [81, 72]. KLEE [10] and SAGE [12] are white-box fuzzers and execute code in a controlled environment. Driller [58] is a hybrid fuzzer that leverages selective concolic execution in a complementary manner. Symbolic execution based fuzzers produce meaningful inputs but tend to be slow [91, 128, 146, 147]. Fuzzers such as LibFuzzer [56], FuzzGen [182] and FUDGE [141] focus on fuzzing libraries, which cannot run as standalone programs, but instead are invoked by other programs.

Several efforts have been made to adopt traditional software fuzzing techniques to test smart contracts. ContractFuzzer [110] for instance, generates inputs based on a list of

input seeds. It also deploys an entire testnet to fuzz transactions, while ConFuzzius is more efficient and solely emulates the EVM. Moreover, ConFuzzius does not rely on user-provided input seeds but instead analyzes the execution traces and uses a constraint solver to generate new values specific to the contract under test. Echidna [176] is a property-based testing tool for smart contracts that leverages grammar-based fuzzing. Echidna requires user-defined predicates in the form of Solidity assertions and does not automatically check for vulnerabilities. Harvey [171] predicts new inputs based on instruction-granularity cost metrics. In contrast, ConFuzzius exploits lightweight symbolic execution when the population fitness does not increase. Further, Harvey fuzzes transaction sequences in a targeted and demand-driven way, assisted by an aggressive mode that directly fuzzes the persistent state of a smart contract. Instead, ConFuzzius relies on the read-after-write dependencies to guide the selection and crossover operators to create meaningful transaction sequences efficiently. ILF [160] is based on imitation learning, which requires a learning phase prior to fuzzing. ILF consists of a neural network that is trained on transactions obtained by running a symbolic execution expert over a broad set of contracts. ConFuzzius does not have the overhead of a learning phase and uses on-demand constraint solving while actively fuzzing the target. Moreover, ILF is limited to the knowledge that it learned during the learning phase, meaning that ILF has issues in getting past program conditions that require inputs that were not part of the learning dataset. ConFuzzius does not have this issue as it learns inputs on-the-fly that are tailored to target contract that is being fuzzed. sFuzz [186] is an AFL based smart contract fuzzer, whereas EthPloit [201] is a fuzzing based smart contract exploit generator. Both sFuzz and EthPloit have been developed concurrently and independently of ConFuzzius. While sFuzz follows a random strategy to create transaction sequences, EthPloit uses static taint analysis on state variables to create meaningful transaction sequences. However, static taint analysis has the disadvantage of being imprecise and analyzing parts that are not executable. Despite sFuzz using a genetic algorithm as ConFuzzius, it employs a different encoding of individuals. Moreover, sFuzz only models block number and timestamp as environmental information. EthPloit, on the other hand, instruments the EVM in a similar way to ConFuzzius. However, EthPloit does not fuzz the size of external code nor contract call return values.

Apart from fuzzing, several other tools based on symbolic execution were proposed to assess the security of smart contracts [51, 122, 120, 102, 158, 116, 181]. MPro [173] combines symbolic execution and data dependency analysis to deal with the scalability issues that symbolic execution tools face when trying to handle the statefulness of smart contracts. MPro has been developed concurrently and independently from ConFuzzius. There are two significant differences to our approach. First, MPro retrieves data dependencies using static analysis and therefore requires source code, whereas ConFuzzius tries to infer data dependencies from bytecode. Second, MPro works in two separate steps, first, it infers data dependencies via static analysis, and then it applies symbolic execution. Con-

FUZZIUS, on the other hand, applies a dynamic approach and infers data dependencies while fuzzing. ETHRACER [115] uses a hybrid approach with a converse strategy by primarily using symbolic execution to test a smart contract and using fuzzing only for producing combinations of transactions to detect vulnerabilities such as transaction order dependency. CONFUZZIUS's fuzzing strategy, compared to ETHRACER, is not entirely random but based on read-after-write dependencies, yielding faster and more efficient combinations.

Besides symbolic execution and fuzzing, other works based on static analysis were proposed to detect smart contract vulnerabilities. ZEUS [112] is a framework for automated verification of smart contracts using abstract interpretation and model checking. SECURIFY [136] uses static analysis based on a contract's dependency graph to extract semantic information about the program bytecode and then checks for violations of safety patterns. Similarly, VANDAL [90] is a framework designed to convert EVM bytecode into semantic logic relations in Datalog, which can then be queried for vulnerabilities.

## 4.6 Conclusion

We presented CONFUZZIUS, the first hybrid fuzzer for smart contracts. It tackles the three main challenges of smart contract testing: *input generation*, *stateful exploration*, and *environmental dependencies*. CONFUZZIUS solves the first challenge by combining evolutionary fuzzing with constraint solving to generate inputs that allow the fuzzer to get past complex path conditions. The second challenge is solved by leveraging data dependency analysis across state variables to generate purposeful transaction sequences. Finally, the third challenge is solved by modeling block related information (e.g., block number) and contract related information (e.g., call return values) as fuzzable inputs. We ran CONFUZZIUS and other state-of-the-art fuzzers and symbolic execution tools for smart contracts against a curated dataset of 128 contracts and a dataset of 21K real-world smart contracts. The results not only show that our hybrid fuzzing approach detects more bugs than existing state-of-the-art tools (up to 23%), but that is also outperforms these tools in terms of code coverage (up to 69%) and that data dependency analysis can boost the detection of bugs (up to 18%).

# Part II

# Attacks

# 5 | Horus

## *Spotting and Analyzing Attacks on Smart Contracts*

*In this chapter, we investigate the analysis and detection of smart contract attacks. In recent years, Ethereum gained tremendously in popularity. As a result, smart contracts have been victims of a number of attacks in the past. In response to these attacks, both academia and industry proposed a plethora of tools to scan smart contracts for vulnerabilities before deploying them on the blockchain. However, most of these tools solely focus on detecting vulnerabilities and not attacks, let alone quantifying or tracing the number of stolen assets. In this chapter, we present* HORUS*, a framework that empowers the automated detection and investigation of smart contract attacks based on logic-driven and graph-driven analysis of transactions.* HORUS *provides quick means to quantify and trace the flow of stolen assets across the Ethereum blockchain. We perform a large-scale analysis of all the smart contracts deployed on Ethereum until May 2020. We identified 1,888 attacked smart contracts and 8,095 adversarial transactions in the wild. Our investigation shows that the number of attacks did not necessarily decrease over the past few years, but for some vulnerabilities remained constant. Finally, we also demonstrate the practicality of our framework via an in-depth analysis on the recent Uniswap and Lendf.me hacks.*

## 5.1 Introduction

In just four years, Ethereum grew from a daily transaction average of 10K in January 2016 to an average of 500K in January 2020 [179]. Such an increase in value and popularity attracts abuse and the lack of a governing authority has led to a "Wild West"-like situation, where several attackers began to exploit vulnerable smart contracts to steal their funds. In the past, several smart contracts hosting tens of millions of USD were victims to attacks (e.g., [47, 84, 80]). Hence, over the past few years a rich corpus of research works and tools have surfaced to identify smart contract vulnerabilities (e.g., [51, 136, 102, 120, 135, 90, 181, 110, 112]). However, most of these tools only focus on analyzing the bytecode of smart contracts and not their transactions or activities. Only a small number leverages transactions to detect attacks (e.g., [165, 175, 198]), whereas the majority either requires the Ethereum client to be modified or large and complex attack detection scripts to be written. Moreover, none of

these tools allow to directly trace stolen assets after their detection.

In this chapter, we introduce HORUS, a framework capable of automatically detecting and analyzing smart contract attacks from historical blockchain data. Besides detecting attacks, the framework also provides means to quantify and trace the flow of stolen assets across Ethereum accounts. The framework replays transactions without modifying the Ethereum client and encodes their execution as logical facts. Attacks are then detected using Datalog queries, making the framework easily extendable to detect new attacks. Stolen funds are traced by loading detected transactions into a graph database and performing transaction graph analysis. Using our framework, we conduct a longitudinal study that spans the entire past Ethereum blockchain history, from August 2015 to May 2020, consisting of over 3 million smart contracts. One of the fundamental research questions we are investigating is whether these years of efforts have yielded visibly fewer attacks in the wild. If the tools proposed herein are effective, one could argue that attacks should have declined over time. To quantify the answer to this question, we start by investigating whether attacks occur continuously, or if they appear sporadically. While most well-known attacks carry significant monetary value, we wonder whether smaller, but ongoing attacks may occur more often and remain rather occluded. In summary, this chapter makes the following contributions:

**Contributions**

- We present the design and implementation of HORUS, a framework that helps identifying smart contract attacks based on a sequence of blockchain transactions using Datalog queries.
- We provide means to quantity stolen funds, including ether as well as tokens, and to trace them across accounts to support behavioral studies of attackers.
- We conduct a longitudinal study on the security of Ethereum smart contracts of 4.5 years, and find 8,095 attacks in the wild, targeting a total of 1,888 vulnerable contracts.
- We demonstrate the practicality of HORUS by performing a forensic analysis on the 2020 Uniswap and Lendf.me hacks.

## 5.2 HORUS

In this section, we provide details on the design and implementation of the HORUS framework. HORUS automates the process of conducting longitudinal studies of attacks on Ethereum smart contracts. The framework has the capability to detect and analyze smart contract attacks from historical data. Moreover, the framework also provides means to trace the flow of stolen assets across Ethereum accounts. The latter is particularly useful for studying the

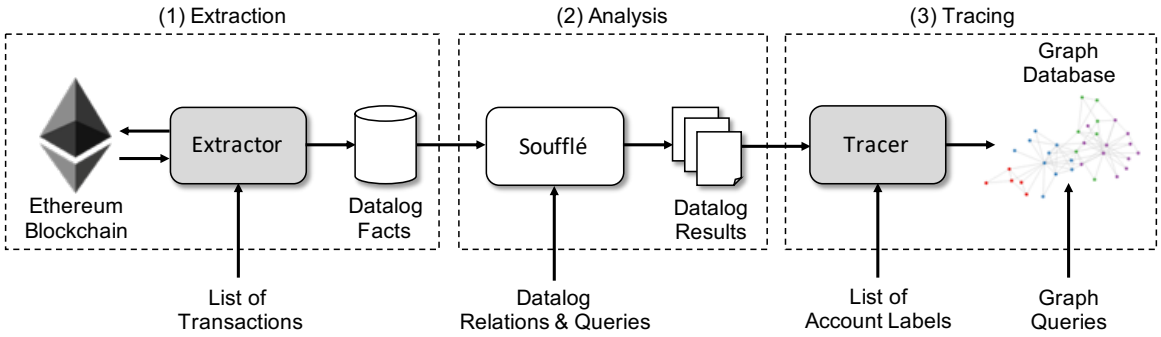



**Figure 5.1:** Architecture of HORUS. Shaded boxes represent custom components, whereas boxes highlighted in white represent off-the-shelf components.

behavior of attackers. Figure 5.1 provides an overview on the architecture of HORUS. The framework is organized as an EAT (extract, analyze, and trace) pipeline consisting of three different stages:

**(1) Extraction:** The extraction stage takes as input a list of transactions from which execution related information is extracted and stored as Datalog facts.

**(2) Analysis:** The analysis stage takes as input a set of Datalog relations and queries, which together identify attacks on the extracted Datalog facts.

**(3) Tracing:** The tracing stage retrieves a list of attacker accounts obtained via the analysis and fetches all transactions related to these accounts (including normal transactions, internal transactions and token transfers). Afterwards, a graph database is created, which captures the flow of funds (both ether and tokens) from and to these accounts. Further, the database can be augmented with a list of labeled accounts to enhance the tracing of stolen assets.

In the following, we describe each of the three pipeline stages in more detail. The entire framework was written in Python using roughly 2,000 lines of code[1].

### 5.2.1 Extraction

The role of the extractor is to request from the Ethereum client the execution trace for a list of transactions and to convert them into logic relations that reflect the semantics of their execution. An execution trace consists of an ordered list of executed EVM instructions. Each record in that list contains information such as the executed opcode, program counter, call stack depth, and current stack values. Unfortunately, execution traces cannot be obtained directly from historical blockchain data, they can only be recorded during contract execution. Fortunately, the Go based Ethereum client (Geth) provides a debug functionality

[1]Code and data are publicly available at: https://github.com/christoftorres/Horus.

```
.decl opcode(step:number, op:Opcode, tx_hash:symbol)
.decl data_flow(step1:number, step2:number, tx_hash:symbol)
.decl arithmetic(step:number, op:Opcode, operand1:Value, operand2:Value,
    arithmetic_result:Value, evm_result:Value)
.decl storage(step:number, op:Opcode, tx_hash:symbol, caller:Address, contract:
    Address, index:Value, value:Value, depth:number)
.decl condition(step:number, tx_hash:symbol)
.decl erc20_transfer(step:number, tx_hash:symbol, contract:Address, from:Address,
    to:Address, value:Value)
.decl call(step:number, tx_hash:symbol, op:Opcode, caller:Address, callee:Address,
     input:symbol, value:Value, depth:number, call_id:number, call_branch:number,
    result:number)
.decl selfdestruct(step:number, tx_hash:symbol, caller:Address, contract:Address,
    destination:Address, value:Value)
.decl block(block_number:number, gas_used:number, gas_limit:number, timestamp:
    number)
.decl transaction(tx_hash:symbol, tx_index:number, block_number:number, from:
    Address, to:Address, input:symbol, gas_used:number, gas_limit:number, status:
    number)
```

**Figure 5.2:** List of Datalog facts extracted by HORUS.

via the `debug_traceTransaction` and `debug_traceBlockByNumber` functions, which gives us the ability to replay the execution of any given past transaction or block and retrieve its execution trace. Execution traces are requested via Remote Procedure Call (RPC). Previous works [165, 218, 175, 200, 198] did not rely on RPC as it is too slow. Instead, they modified Geth to speed up the process of retrieving execution traces. However, this has the limitation that users cannot use Geth's default version, but are required to use a modified version, and changes will need to be carried over every time a new version of Geth is released. Moreover, at the time of writing, none of these works publicly disclosed their modified version of Geth, which not only makes it difficult to reproduce their results, but also to conduct future studies. Therefore, rather than modifying Geth, we decided to improve the speed on the retrieval of execution traces via RPC. We noticed that execution traces contain information that is irrelevant for our analysis. Fortunately, Geth allows us to inject our own execution tracer written in JavaScript [195]. Through this mechanism, we are able to reduce the size of the execution traces and improve execution speed, without actually modifying Geth. For example, our JavaScript code removes the current program counter, the remaining gas and the instruction's gas cost from the execution trace. Moreover, instead of returning a complete snapshot of the entire stack and memory for every executed instruction, our code only returns stack elements and memory slices that are relevant to the executed instruction.

Figure 5.2 shows the list of Datalog facts that our extractor produces by iterating through each of the records of the execution traces and encoding relevant information. While most facts are related to low level EVM operations (e.g., `call`), others are related to high level operations. For example, the `erc20_transfer` fact refers to the ERC-20 token event "Transfer"

that is emitted whenever tokens are transferred, where `contract` denotes the address of the token contract, and `from` and `to`, denote the sender and receiver of the tokens, respectively. It is important to note that this list can easily be modified or extended to support different studies from the one proposed in this chapter by modifying the extractor, analyzer and tracer. Besides using the default types `number` and `symbol`, we also define our own three new types: `Address` for 160-bit values, `Opcode` for the set of EVM opcodes, and `Value` for 256-bit stack values.

**Dynamic Taint Analysis.** The extractor leverages dynamic taint analysis to track the flow of data across instructions. Security experts can then use the `data_flow` fact to check if data flows from one instruction to another. Taint is introduced via sources, then propagated across the execution and finally checked if it flows into sinks. Sources represent instructions that might introduce untrusted data (e.g., `CALLDATALOAD` or `CALLDATACOPY`), whereas sinks represent instructions that are sensitive locations (e.g., `CALL` or `SSTORE`). We implemented our own dynamic taint analysis engine. The engine loops through every executed instruction and checks whether the executed instruction is a source, for which the engine then introduces taint by tagging the affected stack value, memory region or storage location according to the semantics defined in [30]. We implemented the stack using an array structure following LIFO logic. Memory and storage are implemented using a Python dictionary that maps memory and storage addresses to values. Taint propagation is performed at the byte level (see examples in Figure 5.3).

**Execution Order.** Attacks such as the Parity wallets hacks were composed of two transactions being executed in a specific order. To detect such multi-transactional attacks, our framework encodes a total order across multiple transactions via the triplet $o = (b, t, s)$, where $b$ is the block number, $t$ is the transaction index, and $s$ is the execution step. The execution step is a simple counter that is reset at the beginning of the execution of a transaction

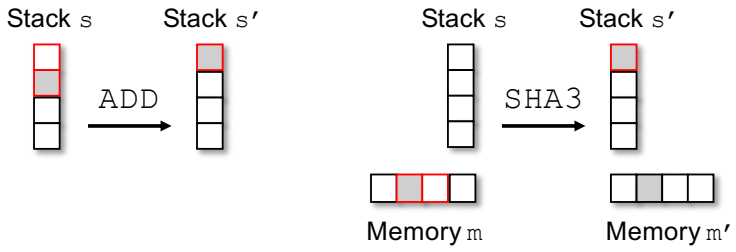



**Figure 5.3:** The example on the left depicts the propagation of taint via the `ADD` instruction, where the result pushed onto stack $s'$ becomes tainted because the second operand on stack $s$ was tainted. The example on the right depicts the propagation of taint via the `SHA3` instruction, where the result pushed onto stack $s'$ becomes tainted because the memory $m$ was tainted.

and its value is incremented after each executed instruction. An execution step is bound to a transaction index, which is on the other hand bound to a block number. As such, our framework is able to precisely identify the execution order of any instruction across multiple transactions and the entire blockchain history.

### 5.2.2 Analysis

The second stage of our pipeline uses a Datalog engine to analyze whether a given list of Datalog relations and queries match any of the previously extracted Datalog facts. These Datalog queries identify adversarial transactions, i.e., malicious transactions that successfully carried out a concrete attack against a smart contract by exploiting a given vulnerability. Our framework uses Soufflé as its Datalog engine. Soufflé compiles Datalog relations and queries into a highly optimized C++ executable [45]. In the following, we provide Datalog queries for detecting reentrancy, Parity wallet hacks, integer overflows, unhandled exceptions and short address attacks. In this chapter, we focus on detecting vulnerabilities that are ranked by the NCC Group as the top 10 smart contract vulnerabilities [105] and for which we can extract the amount of ether or tokens that were either stolen or locked.

**Reentrancy.** Reentrancy occurs whenever a contract calls another contract, and the called contract calls back the original contract (i.e., a re-entrant call) before the state in the original contract has been updated appropriately. We detect reentrancy by identifying cyclic calls originating from the same caller and calling the same callee (see Figure 5.4). We check if two successful `call`s (i.e., result is `1`), share the same transaction `hash`, `caller`, `callee`, `id` and `branch`, where the second call has a higher call `depth` than the first call. Afterwards, we check if there are two `storage` operations with the same call depth as the first call, where the first operation is an `SLOAD` and occurs before the first call, and the second operation is an `SSTORE` and occurs after the second call.

```
Reentrancy(hash, caller, callee, depth2, amount) :-
  storage(step1, "SLOAD", hash, _, caller, index, _, depth1),
  call(step2, hash, _, caller, callee, _,      _, depth1, id, branch, 1),
  call(step3, hash, _, caller, callee, _, amount, depth2, id, branch, 1),
  storage(step4, "SSTORE", hash, _, caller, index, _, depth1),
  depth1 < depth2, step1 < step2, step3 < step4, !match("0", amount).
```

**Figure 5.4:** Datalog query for detecting reentrancy attacks.

**Parity Wallet Hacks.** In this chapter, we focus on detecting the two Parity wallet hacks [84, 80]. Both hacks were due faulty access control implementations that allowed attackers to set themselves as owners, which allowed them to perform critical actions such as the transfer of funds or the destruction of contracts. We detect the first Parity wallet hack by checking

if there exist two transactions $t_1$ and $t_2$, both containing the same sender and receiver, where the first 4 bytes of $t_1$'s input match the function signature of the `initWallet` function (i.e., `e46dcfeb`), and if the first 4 bytes of $t_2$'s input match the function signature of the `execute` function (i.e., `b61d27f6`) (see Figure 5.5). Afterwards, we check whether there is a call, which is part of $t_2$ and where $t_2$ is executed after $t_1$ (i.e., block1 $<$ block2; block1 $=$ block2, index1 $<$ index2).

```
ParityWalletHack1(hash1, hash2, caller, callee, amount) :-
  transaction(hash1, index1, block1, from, to, input1, _, _, 1),
  substr(input1, 0, 8) = "e46dcfeb",
  transaction(hash2, index2, block2, from, to, input2, _, _, 1),
  substr(input2, 0, 8) = "b61d27f6",
  call(_, hash2, "CALL", caller, callee, _, amount, _, 1),
  (block1 < block2; block1 = block2, index1 < index2).
```

**Figure 5.5:** Datalog query for detecting the first Parity wallet hack.

We detect the second Parity wallet hack in a very similar way to the first one, except that in this case we check if $t_2$'s input matches the function signature of the `kill` function (i.e., `cbf0b0c0`) and $t_2$ contains a selfdestruct (see Figure 5.6).

```
ParityWalletHack2(hash1, hash2, contract, destination, amount) :-
  transaction(hash1, index1, block1, from, to, input1, _, _, 1),
  substr(input1, 0, 8) = "e46dcfeb",
  transaction(hash2, index2, block2, from, to, input2, _, _, 1),
  substr(input2, 0, 8) = "cbf0b0c0",
  selfdestruct(_, hash2, _, contract, destination, amount),
  (block1 < block2; block1 = block2, index1 < index2).
```

**Figure 5.6:** Datalog query for detecting the second Parity wallet hack.

**Integer Overflows.** We detect integer overflows by checking if data from `CALLDATALOAD` or `CALLDATACOPY` opcodes flows into an arithmetic operation, where the arithmetic result does not match the result returned by the EVM. Afterwards, we check whether the result of the arithmetic operation flows into an `SSTORE` storage operation and an erc20_transfer occurs, where the amount is one of the two operands used in the arithmetic computation (see Figure 5.7). Please note that in this work, we only focus on detecting integer overflows related to ERC-20 tokens, since token smart contracts have been identified in the past to be frequent victims of integer overflows [126, 127].

```
IntegerOverflow(hash, from, to, amount) :-
  (opcode(step1, "CALLDATALOAD", hash);
   opcode(step1, "CALLDATACOPY", hash)),
  arithmetic(step2, _, operand1, operand2, arithmetic_res, evm_res),
  arithmetic_res != evm_res, (operand1 = amount; operand2 = amount),
  storage(step3, "SSTORE", hash, _, _, _, _, 1),
  data_flow(step1, step2, hash), data_flow(step2, step3, hash),
  erc20_transfer(_, hash, _, from, to, amount), !match("0", amount).
```

**Figure 5.7:** Datalog query for detecting integer overflow attacks.

**Unhandled Exception.** Inner calls executed by smart contracts may fail and by default only the state changes caused by those failed calls are rolled back. It is the responsibility of the developer to check the result of every call and perform proper exception handling. However, many developers forget or decide to ignore the handling of such exceptions, resulting in funds not being transferred to their rightful owners. We detect an unhandled exception by checking whether a `call` with opcode `"CALL"` failed (i.e., result is `0`) with an `amount` larger than zero and where the result was not used in a `condition` (see Figure 5.8).

```
UnhandledException(hash, caller, callee, amount) :-
  call(step, hash, "CALL", caller, callee, _, amount, _, 0),
  !match("0", amount), !used_in_condition(step, hash).
```

**Figure 5.8:** Datalog query for detecting unhandled exceptions.

**Short Address.** The ERC-20 functions `transfer` and `transferFrom` take as input a destination address and a given amount of tokens. During execution the EVM will add trailing zeros to the end of the transaction input if the transaction arguments are not correctly encoded as chunks of 32 bytes, thereby shifting the input bytes to the left by a few zeros, and therefore unwillingly increase the number of tokens to be transferred. However, attackers can exploit this fact by generating addresses that end with trailing zeros and omit these zeros, to then trick another party (e.g., web service) into making a call to `transfer`/`transferFrom` containing the attacker's malformed address. We detect a short address attack by first checking if the first 4 bytes of a `transaction`'s input match either the function signature of `transfer` (i.e., `a9059cbb`) or `transferFrom` (i.e., `23b872dd`). Then, for the function `transfer` we check whether the length of the input is smaller than 68 (i.e., 4 bytes function signature, 32 bytes destination address, and 32 bytes amount), and for the function `transferFrom` we check whether the length of the input is smaller than 100 (i.e., 4 bytes function signature, 32 bytes from address, 32 bytes destination address, and 32 bytes amount), and finally we check if an `erc20_transfer` occurred (see Figure 5.9).

```
ShortAddress(hash, from, to, amount) :-
  transaction(hash, _, _, input, _, _, 1, _),
  (substr(input, 0, 8) = "a9059cbb", strlen(input) / 2 < 68;
   substr(input, 0, 8) = "23b872dd", strlen(input) / 2 < 100),
  erc20_transfer(_, hash, _, from, to, amount), !match("0", amount).
```

**Figure 5.9:** Datalog query for detecting short address attacks.

### 5.2.3 Tracing

The final stage of our pipeline is the tracing of stolen assets, such as ether and tokens, from attacker accounts to labeled accounts (e.g., exchanges). The tracer starts by extracting sender addresses and timestamps from malicious transactions that have been identified via the Datalog analysis. Sender addresses are assumed to be accounts belonging to attackers. Afterwards, the tracer uses Etherscan's API to retrieve for each sender address all its normal transactions, internal transactions and token transfers, and loads them into a Neo4j graph database. We rely on a third-party service such as Etherscan to retrieve normal transactions, internal transactions and token transfers, because a default Ethereum node does not provide this functionality out-of-the-box. Accounts are encoded as vertices and transactions as directed edges between those vertices. We differentiate between three types of accounts: attacker accounts, unlabeled accounts, and labeled accounts. Every account type contains an address. Labeled accounts contain a category (e.g., exchange) and a label (e.g., Kraken 1). We obtain categories and labels from Etherscan's large collection of labeled accounts[2]. We downloaded a total of 5,437 labels belonging to 204 categories.

We differentiate between three different types of transactions: normal transactions, internal transactions, and token transactions. Each transaction type contains a transaction value, transaction hash, and transaction date. Token transactions contain a token name, token symbol and number of decimals. Transactions can be loaded either backwards or forwards. Loading transactions forwards allows us to track where attackers sent their stolen funds to, whereas loading transactions backwards allows us to track where attackers received their funds from. We start with the attacker's account when loading transactions and recursively load transactions for neighboring accounts that are part of the same transaction for up to a given number of hops. We do not load transactions for accounts with more than 1,000 transactions. This is to avoid bloating the graph database with transactions from mixing services, exchanges or gambling smart contracts. Moreover, when loading transactions backwards, we only load transactions that occurred before the timestamp of the attack, whereas when loading transactions forwards, we only load transactions that occurred after the timestamp of the attack. Finally, when all transactions are loaded, security experts can query the graph database using Neo4j's own graph query language called Cypher, to trace the flow of stolen

[2] https://etherscan.io/labelcloud

funds. Evidently, our tracing is only effective up to a certain point, since mixing services and exchanges prevent further tracing. Nonetheless, our tracing is still useful to study whether attackers send their funds to mixers or exchanges and to identify which services are being used and to what extent.

## 5.3 Evaluation

In this section, we demonstrate the scalability and effectiveness of our framework by performing a large-scale analysis of the Ethereum blockchain and comparing our results to those presented in previous works.

**Dataset.** We used the Ethereum ETL framework [162] to retrieve a list of transactions for every smart contract deployed up to block 10M. We collected a total of 697,373,206 transactions and 3,362,876 contracts. The deployment timestamps of the collected contracts range from August 7, 2015, to May 4, 2020. We filtered out contracts without transactions and removed transactions that have a gas limit of 21,000 (i.e., do not execute code). Moreover, similar to [165], we skipped all the transactions that were part of the 2016 denial-of-service attacks, as these incur high execution times [54]. After applying these filters, we ended up with a final dataset of 1,234,197 smart contracts consisting of 371,419,070 transactions. During the extraction phase, HORUS generated roughly 700GB of Datalog facts on the final dataset.

**Experimental Setup.** All experiments were conducted using a machine with 64 GB of memory and an Intel(R) Core(TM) i7-8700 CPU with 12 cores clocked at 3.2 GHz, running 64-bit Ubuntu 18.04.5 LTS. Moreover, we used Geth version 1.9.9, Soufflé version 1.7.1, and Neo4j version 4.0.3.

### 5.3.1 Results

Table 5.1 summarizes our results: we found 1,888 attacked contracts and 8,095 adversarial transactions. From these contracts, 46 were attacked using reentrancy, 600 were attacked during the Parity wallet hacks, 125 were attacked via integer overflows, 1,068 suffered from unhandled exceptions, and 55 were victims of short address attacks. For the Parity wallet hacks, we find that the majority was attacked during the first hack. We also observe that most contracts that are vulnerable to integer overflows were attacked via an integer underflow.

### 5.3.2 Validation

We confirm our framework's correctness by comparing our findings to those reported by previous works for which results were publicly available. Also, we solely compare our finding to

**Table 5.1:** Summary of detected vulnerable contracts and adversarial transactions.

| | Results | | Validation | | |
|---|---|---|---|---|---|
| **Vulnerability** | **Contracts** | **Transactions** | **TP** | **FP** | $p$ |
| Reentrancy | 46 | 2,508 | 45 | 1 | 0.97 |
| Parity Wallet Hacks | 600 | 1,852 | 600 | 0 | 1.00 |
| Parity Wallet Hack 1 | 596 | 1,632 | 596 | 0 | 1.00 |
| Parity Wallet Hack 2 | 238 | 238 | 238 | 0 | 1.00 |
| Integer Overflow | 125 | 443 | 65 | 0 | 1.00 |
| Overflow (*Addition*) | 37 | 139 | 25 | 0 | 1.00 |
| Overflow (*Multiplication*) | 23 | 120 | 20 | 0 | 1.00 |
| Underflow (*Subtraction*) | 104 | 352 | 68 | 0 | 1.00 |
| Unhandled Exception | 1,068 | 3,100 | 100 | 0 | 1.00 |
| Short Address | 55 | 275 | 5 | 0 | 1.00 |
| Total Unique | 1,888 | 8,095 | | | |

works that, similarly to HORUS, focus on detecting attacks rather than vulnerable contracts. In cases where the results were not publicly available, we manually inspected the source code and transactions of flagged contracts using Etherscan. Table 5.1 summarizes the results of our validation in terms of true positives (TP), false positives (FP) and precision ($p$). Overall our framework achieves a high precision of 99.54%.

**Reentrancy.** First, we compare our results to those of SEREUM [165]. The authors reported a total of 16 vulnerable contracts, where 14 are false positives. The true positives include the DAO [41] and the DSEthToken [67] contract, which HORUS has also identified. HORUS has flagged none of the 14 false positives. Next, we compare our results to ÆGIS [157, 180]. HORUS successfully detected the 7 contracts that were reported by ÆGIS. Then, we compare our results to SODA [175]. HORUS identified 25 of the 26 contracts that were flagged as true positives by SODA. We analyzed the remaining contract (`0x59abb8006b30d7357869760d21b4965475198d9d`) and found that it is not vulnerable to reentrancy, which is in line with what other previous works discovered [198]. For the 5 false positives reported by SODA, we detected 3 of them, where two (`0xd4cd7c881f5ceece4917d856ce73f510d7d0769e` and `0x72f60eca0db6811274215694129661151f97982e`) are actual true positives and have been misclassified by SODA. The other one (known as HODLWallet [98]) is indeed a false positive. Afterwards, we compare our results with those of ETHSCOPE [198]. HORUS detected 45 out of the 46 true positives reported by ETHSCOPE. The non-reported contract is the DarkDAO [42], which did not suffer from a reentrancy attack and is, therefore, a false positive. In terms of false positives, HORUS only has one in common with ETHSCOPE, namely the aforementioned HODLWallet contract. The other two false positives that ETHSCOPE reported were correctly identified as true negatives by HORUS. Finally, we compare our results with those of Zhou et al. [203]. HORUS found 22 of the 26 contracts that

have been reported as true positives by Zhou et al. We inspected the remaining 4 contracts and found that they are false positives.

**Parity Wallet Hacks.** For the first Parity wallet hack, we compared our results to those reported by ÆGIS and Zhou et al. ÆGIS reported 3 contracts, which have also been found by HORUS. Next, Zhou et al. reported 622 contracts, of which HORUS found 596. We analyzed the remaining 26 contracts and found that these are false positives. After analyzing their list of transactions, we could not find evidence of the two exploiting transactions, namely `initWallet` and `execute`. For the second Parity wallet hack, we compared our results to those of ÆGIS. HORUS found 238 contracts, of which 236 were also reported by ÆGIS. The remaining two are true positives and have not been identified by ÆGIS.

**Integer Overflow.** We compared our findings to those of Zhou et al. The authors found 50 contracts, whereas we found 125 contracts. HORUS detected 49 of the 50 contracts reported by Zhou et al. We analyzed the undetected contract (`0xa9a8ec071ed0ed5be571396438a046a423a0c206`) and found no evidence of an integer overflow. Besides our comparison with Zhou et al., we also tried to analyze manually the source code of the reported contracts. We were able to obtain the source code for 65 of the 125 reported contracts. Our manual inspection identified that all of the contracts are true positives. They either contained a faulty arithmetic check or no arithmetic check at all.

**Unhandled Exception.** Since none of the previous works analyzed unhandled exceptions, we manually analyzed the source code of the contracts reported by HORUS. However, we limited our validation to a random sample of 100 contracts since manually analyzing 1,068 contracts is infeasible. We find that all of the 100 contracts contained in their source code either a direct call or a function call that did not check the return value. Therefore, we conclude that HORUS reports no false positives on the detection of unhandled exceptions.

**Short Address.** We compared our results to those reported by SODA. SODA detected 726 contracts and 6,599 transactions, whereas HORUS detected 55 contracts and 275 transactions. After further investigation, we found that the contracts and transactions detected by HORUS were also detected by SODA. We also found that SODA reported transactions that failed or where the transferred amount was zero, while HORUS only reported transactions that were successful and where an ERC-20 transfer event was successfully triggered with an amount larger than zero. Moreover, we were able to obtain the source code for 5 of the reported contracts and confirm that the `transfer` or `transferFrom` functions contained inside those contracts do not validate the input length of parameters.

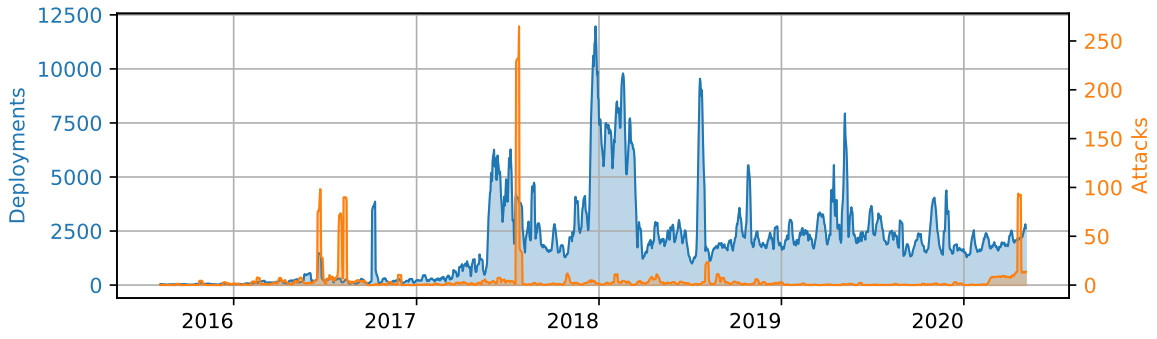



**Figure 5.10:** Weekly average of daily contract deployments and attacks over time.

## 5.4 Analysis

In this section, we demonstrate the practicality of HORUS in detecting and analyzing real-world smart contract attacks via an analysis of our evaluated results and a case study on the recent Uniswap and Lendf.me incidents.

### 5.4.1 Volume and Frequency of Attacks

Figure 5.10 depicts the weekly average of daily attacks in comparison to the weekly average of daily deployments. We note that the peak of weekly deployed contracts was at the end of 2017, and that the largest volume of weekly attacks occurred before this peak. Moreover, most attacks seem to occur in clusters of the same day. We suspect that attackers scan the blockchain for similar vulnerable contracts and exploit them at the same time. The first three spikes in the attacks correspond to the DAO and Parity wallet hacks, whereas the last spike corresponds to the recent Uniswap/Lendf.me hacks.

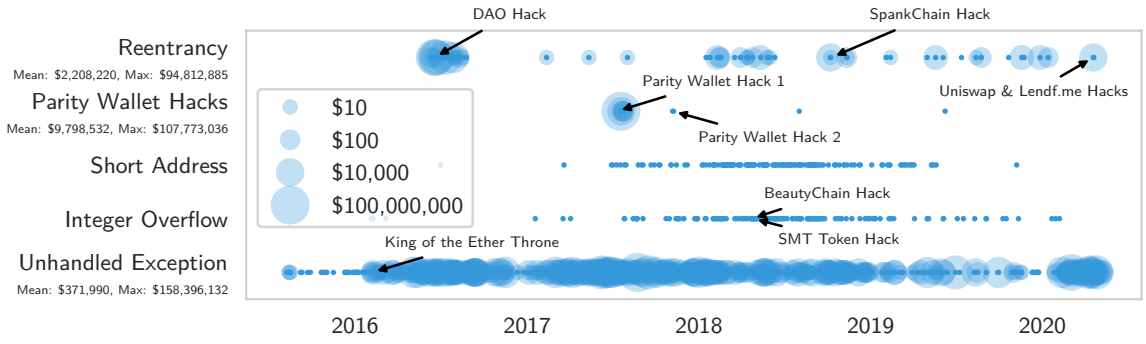



**Figure 5.11:** Volume and frequency of smart contract attacks over time.

Figure 5.11 depicts the occurrences of adversarial transactions per vulnerability type that we measured during our evaluation. While reentrancy attacks seem to occur more sporadically, other types of vulnerabilities such as unhandled exceptions are triggered rather

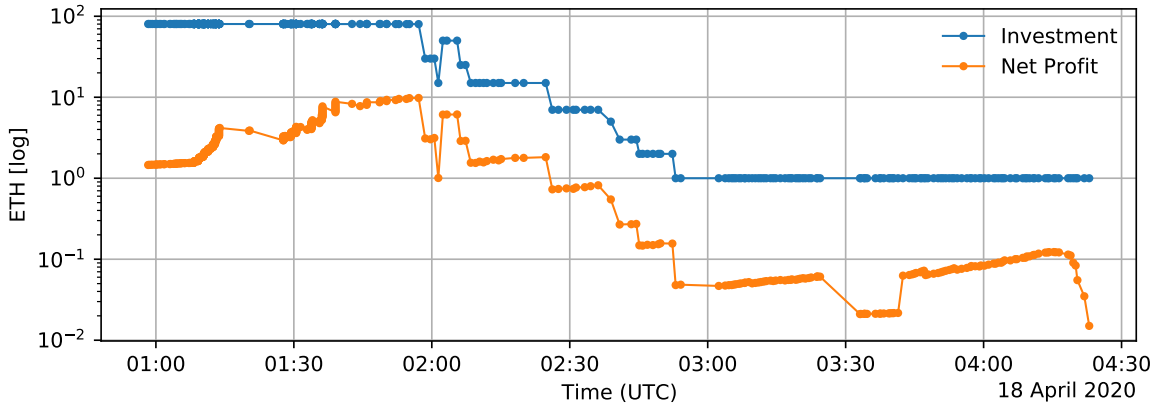



**Figure 5.12:** Invested ETH and net profit made by Uniswap attackers over time.

continuously. Overall, we see that over time less contracts became victims to short address attacks and integer overflows, suggesting that smart contracts have become more secure over the past few years. However, we also see that smart contracts still remain vulnerable to well-known vulnerabilities such as reentrancy and unhandled exceptions, despite automated security tools being available. Figure 5.11 also illustrates for each adversarial transaction the amount of USD that was either stolen (reentrancy and Parity wallet hack 1) or locked (unhandled exception and Parity wallet hack 2). The USD amounts were calculated by multiplying the price of one ether at the time of the attack with the amount of ether extracted via our Datalog query. We do not provide USD amounts for short address attacks and integer overflows, because these attacks involve stolen ERC-20 tokens and we were not able to obtain the historical prices of these tokens. We can see that the DAO hack and the first Parity wallet hack remain the two most devastating attacks in terms of ether stolen, with ether worth 94,812,885 USD and 107,773,036 USD, respectively. We marked well-known incidents such as the DAO hack, or the two Parity wallet hacks for the reader's convenience and to demonstrate that HORUS is able to detect them.

### 5.4.2 Forensic Analysis on Uniswap and Lendf.me Incidents

**Uniswap.** On April 18, 2020, attackers were able to drain a large amount of ether from Uniswap's liquidity pool of ETH-imBTC [154]. They purposely chose the imBTC token as it implements the ERC777 standard, which would allow them to register a callback function and therefore perform a reentrancy attack on Uniswap. The attackers would start by purchasing imBTC tokens for ETH. Afterward, they would exchange half of the purchased imBTC tokens within the same transaction back to ETH. However, the latter would trigger a callback function that the attackers registered before the attack, allowing them to take control and call back the Uniswap contract to exchange the remaining half of imBTC tokens to ETH before the conversion rate was updated. Thus, the attackers could trade the second batch

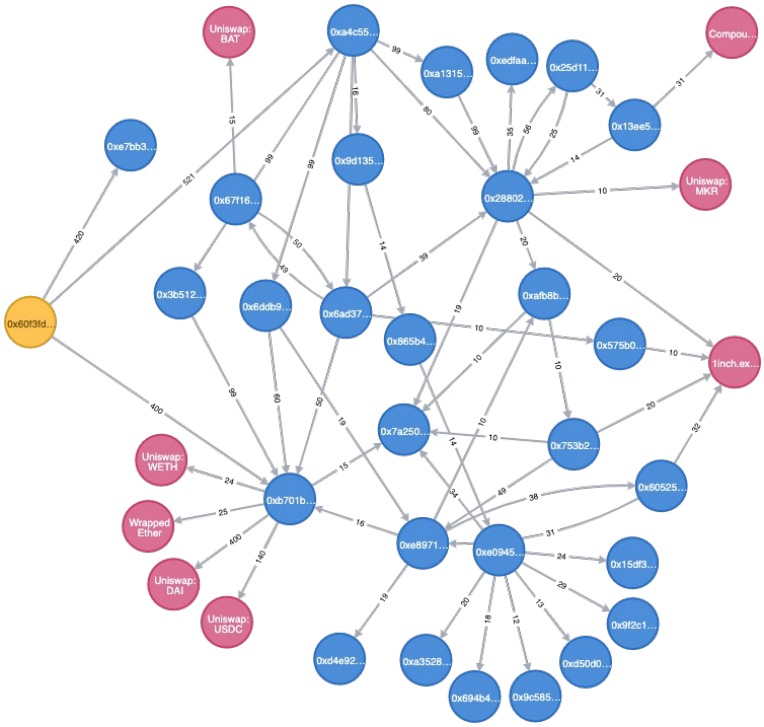



**Figure 5.13:** Transaction graph of Uniswap incident with normal transactions loaded forwards for up to 5 hops. Yellow node highlights Uniswap attacker whereas pink nodes highlight exchanges.

of imBTC tokens at a more profitable conversion rate. Interestingly, this vulnerability was known to Uniswap and was publicly disclosed precisely a year before the attack [148].

We used HORUS to extract and analyze all the transactions mined on that day, and identified a total of 525 transactions performing reentrancy attacks against Uniswap with an accumulated profit of 1,278 ETH (232,239.46 USD). The attack began at 00:58:19 UTC and ended roughly 3.5 hours later at 04:22:58 UTC. Figure 5.12 depicts a timeline of the attack, showing the amount of ether that the attackers invested and the net profit they made per transaction. We see that the net profit goes down over time. The highest profit made for a single transaction was roughly 9.79 ETH (1,778.72 USD), while the lowest profit was 0.01 ETH (2.73 USD). The attackers began their attack by purchasing tokens for roughly 80 ETH and went over time down to 1 ETH. Moreover, we see that the profit was mostly tied to the amount of ether that the attackers were investing (i.e., using to purchase imBTC tokens). However, we also see that sometimes there were some fluctuations, where the attackers were making more profit while they would invest the same amount of ether. This is probably due to other participants trading imBTC on Uniswap during the attack and therefore influencing the exchange rates. In the last step, we traced the entire ether flow from the attackers account for up to 5 hops using HORUS's tracing capabilities (see Figure 5.13). Our transaction graph analysis reveals that the attackers exchanged roughly 702 ETH (55%

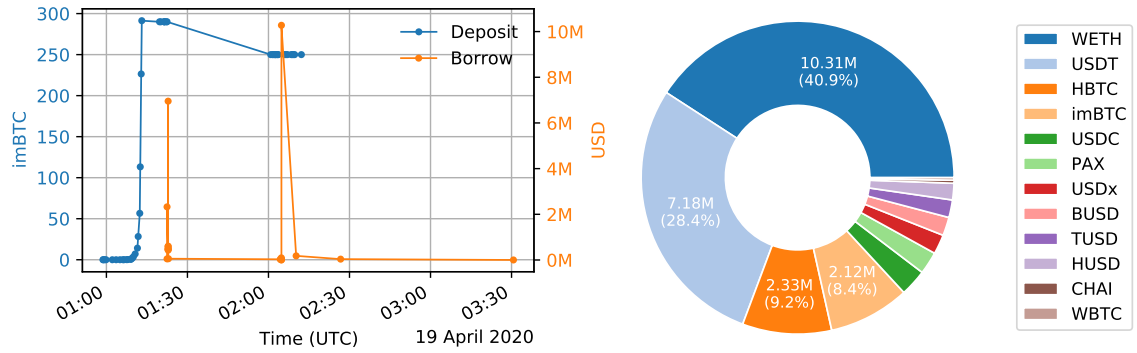



**Figure 5.14:** Deposited and borrowed tokens by Lendf.me attackers over time.

of the stolen funds) for tokens on different exchanges: 589 ETH on Uniswap for WETH, DAI, USDC, BAT, and MKR, 31 ETH on Compound, and 82 ETH on 1inch.exchange. The latter is of particular interest for law enforcement agencies as 1inch.exchange keeps track of IP addresses of transactions performed over their platform [192], which can be useful in deanonymizing the attackers.

**Lendf.me.** On April 19, 2020, attackers were able to drain all tokens from Lendf.me's liquidity pools [153]. Similar to the Uniswap hack, the attackers exploited the fact that Lendf.me was trading imBTC and could register a callback function to perform a reentrancy attack. The attackers would start by depositing $x$ amount of imBTC tokens into Lendf.me's liquidity pool. Next, still within the same transaction, they would deposit another amount $y$, however, this time triggering the callback function registered by the attackers, which would withdraw the previously deposited $x$ tokens from Lendf.me. By the end of the transaction, the imBTC balance of the attackers on the imBTC token contract would be $x - y$, but the imBTC balance on the Lendf.me contract would be $x + y$, thereby increasing their imBTC balance on Lendf.me by $x$ without actually depositing it. Similar to Uniswap, the issue here is that the user's balance is only updated after the transfer of tokens, thus the update is based on data before the transfer and therefore ignoring any updates made in between.

Using HORUS, we extracted and analyzed all the transactions mined on that day. We identified a total of 46 transactions performing reentrancy attacks against Lendf.me, and 19 transactions using the stolen imBTC tokens to borrow other tokens. Figure 5.14 shows on the left the amount of imBTC tokens that the attackers deposited during the attack and the amount of USD that the attackers made by borrowing other tokens. The right-hand side of Figure 5.14 depicts the number of tokens in USD that the attackers borrowed from Lendf.me. The attackers borrowed from 12 different tokens, worth together 25,244,120.74 USD, where 10.31M USD are only from borrowing WETH. The attackers launched their attack at 00:58:43 UTC and stopped 2 hours later at 02:12:11 UTC. They started depositing low amounts of imBTC and increased their amounts over time up to 291.35 imBTC. The borrowing started

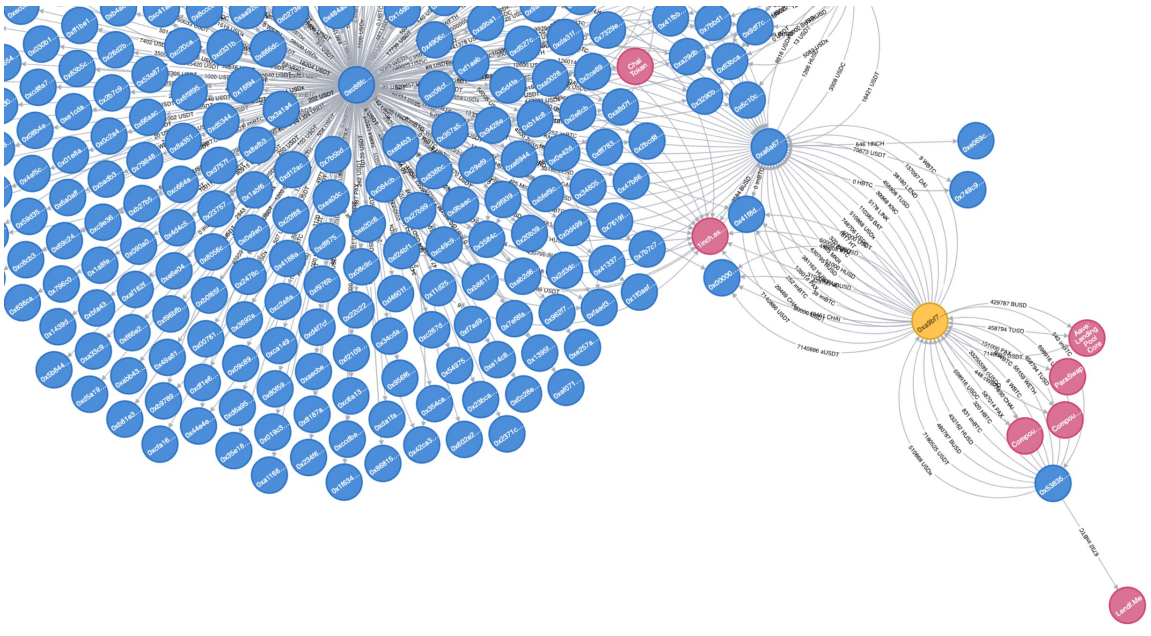


**Figure 5.15:** Transaction graph of Lendf.me incident with token transfers loaded forwards for up to 3 hops. Yellow node highlights Lendf.me attacker whereas pink nodes highlight exchanges.

at 01:22:27 UTC and ended at 03:30:42 UTC. Finally, we used HORUS to trace the flow of tokens from the attackers account for up to 3 hops (see Figure 5.15). We found that the attackers initially traded some parts of the stolen tokens for other tokens on ParaSwap, Compound, Aave, and 1inch.exchange. However, at 14:16:52 UTC, thus about 10 hours later, the attackers started sending all the stolen tokens back to Lendf.me's admin account (`0xa6a6783828ab3e4a9db54302bc01c4ca73f17efb`). Lendf.me then moved all the tokens into a recovery account (`0xc88fcc12f400a0a2cebe87110dcde0dafd29f148`) where users could then reclaim their tokens.

## 5.5 Related Work

Researchers proposed a number of tools to detect smart contract vulnerabilities via static analysis. Luu et al. [51] proposed OYENTE, the first symbolic execution tool for smart contracts. Other tools such as OSIRIS [102], combine symbolic execution and taint analysis to detect integer bugs. MYTHRIL [120] uses a mix of symbolic execution and control-flow checking. MAIAN [122] employs inter-procedural symbolic execution. TEETHER [116] automatically generates exploits for smart contracts. HONEYBADGER [158] performs symbolic execution to detect honeypots. However, symbolic execution is often unable to explore all program states, making it generally unsound. Formal verification tools were proposed [117, 170], together with a formal definition of the EVM [107]. ETHBMC [181] uses bounded model checking to detect vulnerabilities, whereas ETHOR [193] uses reachability analy-

sis. ZEUS [112] verifies the correctness of smart contracts using abstract interpretation and model checking. SMARTCHECK [135] checks Solidity source code against XPath patterns. VERISMART [194] leverages counter example-based inductive synthesis to detect arithmetic bugs. SECURIFY [136] extracts semantic information from the dependency graph to check for compliance and violation patterns using Datalog. VANDAL [90] converts EVM bytecode to semantic logic relations and checks them against Datalog queries. The main difference between these works and ours, is that they analyze the bytecode of smart contracts, whereas we analyze the execution of transactions.

Although less apparent, a number of dynamic approaches have also been proposed. ECFCHECKER [69] enables the runtime detection of reentrancy attacks via a modified EVM. SEREUM [165] proposes a modified EVM to protect deployed smart contracts against reentrancy attacks. ÆGIS [157, 180] presents a smart contract and a DSL to protect against all kinds of runtime attacks. SODA [175] uses a modified Ethereum client to inject custom modules for the online detection of malicious transactions. Perez et al. [218] use Datalog to study the transactions of vulnerable smart contracts that have been detected by previous works. ETHSCOPE [198] loads historical data into an Elasticsearch database and adds dynamic taint analysis to the client to analyze transactions. Zhou et al. [203] study attacks and defenses by encoding transactional information as action trees and result graphs. TXSPECTOR [200] is a concurrent work to ours and adopts the Datalog facts proposed by VANDAL. However, these facts were designed to analyze bytecode and do not allow to detect multi-transactional attacks. In contrast to these works, our work does not modify the Ethereum client. Instead, we dynamically inject our custom tracer into the client. We also provide a new set of Datalog facts that allow to check for multi-transactional attacks and describe data flows between instructions via dynamic taint analysis. Finally, none of the aforementioned tools provide means to trace stolen assets across the Ethereum blockchain.

## 5.6 Conclusion

In this chapter, we presented the design and implementation of an extensible framework called HORUS, for carrying out longitudinal studies on the detection, analysis, and tracing of smart contract attacks. We analyzed transactions from August 2015 to May 2020 and identified 8,095 attacks as well as 1,888 vulnerable contracts. Our analysis revealed that the number of attacks seem to have decreased for attacks such as integer overflows, whereas the number for unhandled exceptions and reentrancy attacks still seem to remain constant despite an abundance of new smart contract security tools. Finally, we also presented an in-depth analysis on the 2020 Uniswap and Lendf.me incidents and demonstrate the practicality of HORUS in performing post-mortem analyses.

# 6 | HoneyBadger

## *Demystifying Smart Contract Honeypots*

*In this chapter, we investigate the emergence of smart contract honeypots. In the past few years, several smart contracts have been exploited by attackers. However, a new trend towards a more proactive approach seems to be on the rise, where attackers do not search for vulnerable contracts anymore. Instead, they try to lure their victims into traps by deploying seemingly vulnerable contracts that contain hidden traps. This new type of contracts is commonly referred to as* honeypots*. In this chapter, we present the first systematic analysis of honeypot smart contracts, by investigating their prevalence, behavior and impact on the Ethereum blockchain. We develop a taxonomy of honeypot techniques and use this to build* HONEYBADGER *– a tool that employs symbolic execution and well defined heuristics to expose honeypots. We perform a large-scale analysis on more than 2 million smart contracts and identify 690 honeypot smart contracts as well as 240 victims in the wild, with an accumulated profit of more than 90,000 USD for the honeypot creators. Finally, our manual validation shows that 87% of the reported contracts are indeed honeypots.*

## 6.1 Introduction

As Ethereum grows and becomes more valuable, attackers also become more incentivized to find and exploit vulnerable contracts. In fact, Ethereum already faced several devastating attacks on vulnerable smart contracts. The most prominent ones being the DAO hack in 2016 [57] and the Parity Wallet hack in 2017 [84], together causing a loss of over 400M USD. In response to these attacks, academia proposed a plethora of different tools that allow to scan contracts for vulnerabilities, prior to deploying them on the blockchain (see e.g., [51, 120, 102]). Unfortunately, these tools may also be used by attackers in order to easily find vulnerable contracts and exploit them. This potentially enables attackers to follow a reactive approach by actively scanning the blockchain for vulnerable contracts. Alternatively, attackers could follow a more proactive approach by luring their victims into traps. In other words: *Why should I spend my time on looking for victims, if I can just let the victims come to me?* This new type of fraud has been introduced by the community as *"honeypots"* (see e.g., [131, 132]). Honeypots are smart contracts that appear to have an obvious flaw in their

design, which allows an arbitrary user to drain ether from the contract, given that the user transfers a priori a certain amount of ether to the contract. However, once the user tries to exploit this apparent vulnerability, a second, yet unforeseen, trapdoor unfolds which prevents the draining of ether to succeed. The idea is that the user solely focuses on the apparent vulnerability and does not consider the possibility that a second vulnerability might be hidden in the contract. Similar to other types of fraud, honeypots work because human beings are often easily manipulated. People are not always capable of quantifying risk against their own greed and presumptions.

In this chapter, we investigate the prevalence of such honeypot smart contracts in Ethereum. To the best of our knowledge, this is the first work to provide an in depth analysis on the inner workings of this new type of fraud. Moreover, we introduce HONEYBADGER – a tool that uses a combination of symbolic execution and precise heuristics to automatically detect various types of honeypots. Using HONEYBADGER, we are able to provide interesting insights on the plethora, anatomy, and popularity of honeypots that are currently deployed on the Ethereum blockchain. Finally, we investigate whether honeypots are profitable and discuss their effectiveness. In summary, this chapter makes the following contributions:

**Contributions**

- We conduct the first systematic analysis of an emerging new type of fraud in Ethereum: *honeypots*.
- We identify common techniques used by honeypots and organize them in a taxonomy.
- We present HONEYBADGER, a tool based on symbolic execution that automatically detects honeypots in Ethereum smart contracts.
- We analyze 2 million smart contracts and confirm the prevalence of at least 690 honeypots in the wild.
- We discover that 240 users already became victims of honeypots, with an accumulated loss of over 90,000 USD for the honeypot creators.

## 6.2 Ethereum Honeypots

In this section, we provide a general definition of a honeypot and introduce our taxonomy of honeypots.

### 6.2.1 Honeypots

**Definition 1** (Honeypot)**.** *A honeypot is a smart contract that pretends to leak its funds to an arbitrary user (victim), provided that the user sends additional funds to it. However, the*

*funds provided by the user will be trapped and only the honeypot creator (attacker) will be able to retrieve them.*

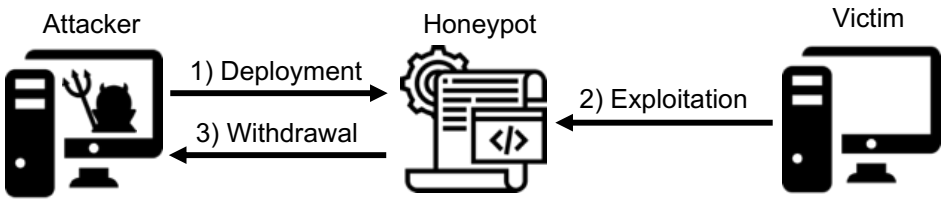



**Figure 6.1:** Actors and phases of a honeypot smart contract.

Figure 6.1 depicts the different actors and phases of a honeypot. A honeypot generally operates in three phases:

(1) The attacker deploys a seemingly vulnerable contract and places a bait in the form of funds;

(2) The victim attempts to exploit the contract by transferring at least the required amount of funds and fails;

(3) The attacker withdraws the bait together with the funds that the victim lost in the attempt of exploitation.

An attacker does not require special capabilities to set up a honeypot. In fact, an attacker has the same capabilities as a regular Ethereum user. He or she solely requires the necessary funds to deploy the smart contract and to place a bait.

### 6.2.2 Taxonomy of Honeypots

We grasped public sources available on the Internet, in order to have a first glimpse at the inner workings of honeypots [139, 118, 131, 130, 132]. We were able to collect a total of 24 honeypots (see Table 6.1) and distill 8 different honeypot techniques. Table 6.1 presents the list of 24 honeypots that have been collected from public sources available on the Internet. We organize the different techniques in a taxonomy (see Table 6.2), whose purpose is twofold: (i) as a reference for users in order to avoid common honeypots in Ethereum; (ii) as a guide for researchers to foster the development of methods for the detection of fraudulent smart contracts. We group the different techniques into three different classes, according to the level on which they operate:

(1) *Ethereum Virtual Machine*

(2) *Solidity Compiler*

(3) *Etherscan Blockchain Explorer*

**Table 6.1:** List of publicly available honeypots on the Internet [139, 118, 131, 130, 132].

| Contract Name | Contract Address | Technique |
|---|---|---|
| **Ethereum Virtual Machine** | | |
| MultiplicatorX3 | 0x5aa88d2901c68fda244f1d0584400368d2c8e739 | Balance Disorder |
| PinCodeEtherStorage | 0x35c3034556b81132e682db2f879e6f30721b847c | Balance Disorder |
| **Solidity Compiler** | | |
| TestBank | 0x70c01853e4430cae353c9a7ae232a6a95f6cafd9 | Inheritance Disorder |
| KingOfTheHill | 0x4dc76cfc65b14b3fd83c8bc8b895482f3cbc150a | Inheritance Disorder |
| RichestTakeAll | 0xe65c53087e1a40b7c53b9a0ea3c2562ae2dfeb24 | Inheritance Disorder |
| ICO_Hold | 0x4ba0d338a7c41cc12778e0a2fa6df2361e8d8465 | Inheritance Disorder |
| TerrionFund | 0x33685492a20234101b553d2a429ae8a6bf202e18 | Inheritance Disorder |
| DividendDistributorv3 | 0x858c9eaf3ace37d2bedb4a1eb6b8805ffe801bba | Skip Empty String Literal |
| For_Test | 0x2ecf8d1f46dd3c2098de9352683444a0b69eb229 | Type Deduction Overflow |
| Test1 | 0x791d0463b8813b827807a36852e4778be01b704e | Type Deduction Overflow |
| CryptoRoulette | 0x94602b0e2512ddad62a935763bf1277c973b2758 | Uninitialised Struct |
| OpenAddressLottery | 0xd1915a2bcc4b77794d64c4e483e43444193373fa | Uninitialised Struct |
| GuessNumber | 0x559cc6564ef51bd1ad9fbe752c9455cb6fb7feb1 | Uninitialised Struct |
| **Etherscan Blockchain Explorer** | | |
| TestToken | 0x3d8a10ce3228cb428cb56baa058d4432464ea25d | Hidden Transfer |
| WhaleGiveaway1 | 0x7a4349a749e59a5736efb7826ee3496a2dfd5489 | Hidden Transfer |
| Gift_1_ETH | 0xd8993f49f372bb014fb088eabec95cfdc795cbf6 | Hidden State Update |
| NEW_YEARS_GIFT | 0x13c547ff0888a0a876e6f1304eaefe9e6e06fc4b | Hidden State Update |
| G_GAME | 0x3caf97b4d97276d75185aaf1dcf3a2a8755afe27 | Hidden State Update |
| IFYKRYGE | 0x1237b26652eebf1cb8f59e07e07101c0df4f60f6 | Hidden State Update |
| EtherBet | 0x3c3f481950fa627bb9f39a04bccdc88f4130795b | Hidden State Update |
| Private_Bank | 0xd116d1349c1382b0b302086a4e4219ae4f8634ff | Straw Man Contract |
| firstTest | 0x42db5bfe8828f12f164586af8a992b3a7b038164 | Straw Man Contract |
| TransferReg | 0x62d5c4a317b93085697cfb1c775be4398df0678c | Straw Man Contract |
| testBank | 0x477d1ee2f953a2f85dbecbcb371c2613809ea452 | Straw Man Contract |

The first class tricks users by making use of the unusual behavior of the EVM. Although the EVM follows a strict and publicly known set of rules, users can still be misled or confused by devious smart contract implementations that suggest a non-conforming behavior. The second class relates to honeypots that benefit from issues that are introduced by the Solidity compiler. While some compiler issues are well known, others still remain undocumented and might go unnoticed if a user does not analyze the smart contract carefully or does not test it under real-world conditions. The final and third class takes advantage of issues that are related to the limited information displayed on Etherscan's website. Etherscan is per-

**Table 6.2:** A taxonomy of honeypot techniques in Ethereum smart contracts.

| Level | Technique |
|---|---|
| Ethereum Virtual Machine | Balance Disorder |
| Solidity Compiler | Inheritance Disorder<br>Skip Empty String Literal<br>Type Deduction Overflow<br>Uninitialized Struct |
| Etherscan Blockchain Explorer | Hidden State Update<br>Hidden Transfer<br>Straw Man Contract |

haps the most prominent Ethereum blockchain explorer and many users fully trust the data displayed therein. In the following, we explain each honeypot technique through a simplified example. We also assume that: 1) the attacker has placed a bait in form of ether into the smart contract, as an incentive for users to try to exploit the contract; 2) the attacker has a way of retrieving the amount of ether contained in the honeypot.

### *Ethereum Virtual Machine*

**Balance Disorder.** Every smart contract in Ethereum possesses a balance. The contract in Figure 6.2 depicts an example of a honeypot that makes use of a technique that we denote as *balance disorder*. The function `multiplicate` suggests that the balance of the contract (`this.balance`) and the value included in the transaction to this function call (`msg.value`) are transferred to an arbitrary address, if the caller of this function includes a value that is higher than or equal to the current balance of the smart contract. Hence, a naive user will believe that all that he or she needs to do, is to call this function with a value that is higher or equal to the current balance, and that in return he or she will obtain the "invested" value plus the balance contained in the contract. However, if a user tries to do so, he or she will quickly realize that line 5 is not executed because the condition at line 4 does not hold. The reason for this is that the balance is already incremented with the transaction value, before the actual execution of the smart contract takes place. It is worth noting that: 1) the condition at line 4 can be satisfied if the current balance of the contract is zero, but then the user does

```
1 contract MultiplicatorX3 {
2   ...
3   function multiplicate(address adr) payable {
4     if (msg.value >= this.balance)
5       adr.transfer(this.balance + msg.value);
6   }
7 }
```

**Figure 6.2:** An example of a balance disorder honeypot.

```
1  contract Ownable {
2    address owner = msg.sender;
3    modifier onlyOwner {
4      require(msg.sender == owner);
5      _;
6    }
7  }
8  contract KingOfTheHill is Ownable {
9    address public owner;
10   ...
11   function() public payable {
12     if(msg.value > jackpot) owner = msg.sender;
13     jackpot += msg.value;
14   }
15   function takeAll() public onlyOwner {
16     msg.sender.transfer(this.balance);
17     jackpot = 0;
18   }
19 }
```

**Figure 6.3:** An example of an inheritance disorder honeypot.

not have an incentive to exploit the contract; 2) the addition `this.balance+msg.value` at line 5, solely serves the purpose of making the user further believe that the balance is updated only after the execution.

### *Solidity Compiler*

**Inheritance Disorder.** Solidity supports inheritance via the `is` keyword. When a contract inherits from multiple contracts, only a single contract is created on the blockchain, and the code from all the base contracts is copied into the created contract. Figure 6.3 shows an example of a honeypot that makes use of a technique that we denote as *inheritance disorder*. At first glance, there seems to be nothing special about this code, we have a contract `KingOfTheHill` that inherits from the contract `Ownable`. We notice two things though: 1) the function `takeAll` solely allows the address stored in variable `owner` to withdraw the contract's balance; 2) the `owner` variable can be modified by calling the fallback function with a message value that is greater than the current jackpot (line 12). Now, if a user tries to call the function in order to set themself as the owner, the transaction succeeds. However, if he or she afterwards tries to withdraw the balance, the transaction fails. The reason for this is that the variable `owner`, declared at line 9, is not the same as the variable that is declared at line 2. We would assume that the `owner` at line 9 would be overwritten by the one at line 2, but this is not the case. The Solidity compiler will treat the two variables as distinct variables and thus writing to `owner` at line 9 will not result in modifying the owner defined in the contract `Ownable`.

**Skip Empty String Literal.** The contract illustrated in Figure 6.4 allows a user to place

an investment by sending a minimum amount of ether to the contract's function `invest`. Investors may withdraw their investment by calling the function `divest`. Now, if we have a closer look at the code, we realize that there is nothing that prohibits the investor from divesting an amount that is greater than the originally invested amount. Thus a naive user is led to believe that the function `divest` can be exploited. However, this contract contains a bug known as *skip empty string literal*[1]. The empty string literal that is given as an argument to the function `loggedTransfer` (line 14), is skipped by the encoder of the Solidity compiler. This has the effect that the encoding of all arguments following this argument are shifted to the left by 32 bytes and thus the function call argument `msg` receives the value of `target`, whereas `target` is given the value of `currentOwner`, and finally `currentOwner` receives the default value zero. Thus, in the end the function `loggedTransfer` performs a transfer to *currentOwner* instead of *target*, essentially diverting all attempts to divest from the contract to transfers to the owner. A user trying to use the smart contract's apparent vulnerability thereby effectively just transfers the investment to the contract owner.

```
1  contract DividendDistributorv3 {
2    ...
3    function loggedTransfer(uint amount,bytes32 msg,address target,address currentOwner){
4      if (!target.call.value(amount)()) throw;
5      Transfer(amount, msg, target, currentOwner);
6    }
7    function invest() public payable {
8      if (msg.value >= minInvestment)
9        investors[msg.sender].investment += msg.value;
10   }
11   function divest(uint amount) public {
12     if (investors[msg.sender].investment == 0 || amount == 0) throw;
13       investors[msg.sender].investment - =amount;
14       this.loggedTransfer(amount, "", msg.sender, owner);
15   }
16 }
```

**Figure 6.4:** An example of a skip empty string literal honeypot.

**Type Deduction Overflow.** In Solidity, when declaring a variable as type `var`, the compiler uses type deduction to automatically infer the smallest possible type from the first expression that is assigned to the variable. The contract in Figure 6.5 depicts an example of a honeypot that makes use of a technique that we denote as *type deduction overflow*. At first, the contract suggests that a user will be able to double the investment. However, since the type is only deduced from the first assignment, the loop at line 7 will be infinite. Variable `i` will have the type `uint8` and the highest value of this type is 255, which is smaller than 2 *

[1] https://github.com/ethereum/solidity/blob/develop/docs/bugs.json

`msg.value`[2]. Therefore, the loop's halting condition will never be reached. Nevertheless, the loop can still be stopped, if the variable `multi` is smaller than `amountToTransfer`. This is possible, since `amountToTransfer` is assigned the value of `multi`, which eventually will be smaller than `amountToTransfer` due to an integer overflow happening at line 8, where `i` is multiplied by 2. Once the loop exits, the contract performs a value transfer back to the caller, although with an amount that will be at most 255 wei (smallest sub-denomination of ether, where 1 ether $= 10^{18}$ wei) and therefore far less than the value the user originally invested.

```
1  contract For_Test {
2    ...
3    function Test() payable public {
4      if (msg.value > 0.1 ether) {
5        uint256 multi = 0;
6        uint256 amountToTransfer = 0;
7        for (var i = 0; i < 2 * msg.value; i++) {
8          multi = i * 2;
9          if (multi < amountToTransfer) {
10           break;
11           amountToTransfer = multi;
12       }
13       msg.sender.transfer(amountToTransfer);
14     }
15   }
16 }
```

**Figure 6.5:** An example of a type deduction overflow honeypot.

**Uninitialized Struct.** Solidity provides means to define new data types in the form of structs. They combine several named variables under one variable and are the basic foundation for more complex data structures in Solidity. An example of an *uninitialized struct* honeypot is given in Figure 6.6. In order to withdraw the contract's balance, the contract requires a user to place a minimum bet and guess a random number that is stored in the contract. However, any user can easily obtain the value of the random number, since every data stored on the blockchain is publicly available. The first thought suggests that the contract creator simply made a common mistake by assuming that variables declared as `private` are secret. An innocent user simply reads the random number from the blockchain and calls the function `guessNumber` by placing a bet and providing the correct number. Afterwards, the contract creates a struct that seems to track the participation of the user. However, the struct is not properly initialized via the `new` keyword. As a result, the Solidity compiler maps the storage location of the first variable contained in the struct (`player`) to the storage location of the first variable contained in the contract (`randomNumber`), thereby overwriting the random number

[2] 2 * 0.1 ether = 2 * $10^{17}$ wei

with the address of the caller and thus making the condition at line 14 fail. It is worth noting that the honeypot creator is aware that a user might try to guess the overwritten value. The creator therefore limits the number to be between 1 and 10 (line 10), which drastically reduces the chances of the user generating an address that fulfills this condition.

```
1  contract GuessNumber {
2    uint private randomNumber = uint256(keccak256(now)) % 10 + 1;
3    uint public lastPlayed;
4    uint public minBet = 0.1 ether;
5    struct GuessHistory {
6      address player;
7      uint256 number;
8    }
9    function guessNumber(uint256 _number) payable{
10     require(msg.value >= minBet && _number <= 10);
11     GuessHistory guessHistory;
12     guessHistory.player = msg.sender;
13     guessHistory.number = _number;
14     if (_number == randomNumber)
15       msg.sender.transfer(this.balance);
16     lastPlayed = now;
17   }
18 }
```

**Figure 6.6:** An example of an uninitialized struct honeypot.

### *Etherscan Blockchain Explorer*

**Hidden State Update.** In addition to normal transactions, Etherscan also displays so-called *internal messages*, which are transactions that originate from other contracts and not from user accounts. However, for usability purposes, Etherscan does not display internal messages that include an empty transaction value. The contract in Figure 6.7 is an example of a honeypot technique that we denote as *hidden state update*. In this example, the balance is transferred to whoever can guess the correct value that has been used to compute the stored hash. A naive user will assume that `passHasBeenSet` is set to `false` and will try to call the unprotected `SetPass` function, which allows to rewrite the hash with a known value, given that least 1 ether is transferred to the contract. When analysing the internal messages on Etherscan, the user will not find any evidence of a call to the `PassHasBeenSet` function and therefore assume that `passHasBeenSet` is set to `false`. However, the filtering performed by Etherscan can be misused by the honeypot creator in order to silently update the state of the variable `passHasBeenSet`, by calling the function `PassHasBeenSet` from another contract and using an empty transaction value. Thus, by just looking at the internal messages displayed on Etherscan, unaware users will believe that the variable is set to `false` and confidently

transfer ether to the `SetPass` function.

```
1  contract Gift_1_ETH {
2    bool passHasBeenSet = false;
3    ...
4    function SetPass(bytes32 hash) payable {
5      if (!passHasBeenSet && (msg.value >= 1ether))
6        hashPass = hash;
7    }
8    function GetGift(bytes pass) returns(bytes32) {
9      if (hashPass == sha3(pass))
10       msg.sender.transfer(this.balance);
11     return sha3(pass);
12   }
13   function PassHasBeenSet(bytes32 hash) {
14     if (hash == hashPass) passHasBeenSet = true;
15   }
16 }
```

**Figure 6.7:** An example of a hidden state update honeypot.

**Hidden Transfer.** Etherscan provides a web interface that displays the source code of a validated smart contract. Validated means that the provided source code has successfully been compiled to the associated bytecode. For quite a while, Etherscan presented the source code within an HTML `textarea` element, where larger lines of code would only be displayed up to a certain width. Thus, the rest of the line of code would be hidden and solely visible by scrolling horizontally. The contract in Figure 6.8 takes advantage of this "feature" by introducing, at line 4 in function `withdrawAll`, a long sequence of white spaces, effectively hiding the code that follows. The hidden code throws, if the caller of the function is not the owner and thereby prevents the subsequent balance transfer to any caller of the function. Also note the check at line 4, where the block number must be greater than 5,040,270. This ensures that the honeypot solely steals funds when deployed on the main network. Since the block numbers on the test networks are smaller, testing this contract on a such a network

```
1  contract TestToken {
2    ...
3    function withdrawAll() payable {
4      require(0.5 ether < total);/*_____________________________________________________*/
           if (block.number > 5040270) {if (_owner == msg.sender){_owner.transfer(this.
           balance);} else {throw;}}
5      msg.sender.transfer(this.balance);
6    }
7  }
```

**Figure 6.8:** An example of a hidden transfer honeypot.

```
1  contract Private_Bank {
2    ...
3    function Private_Bank(address _log) {
4      TransferLog = Log(_log);
5    }
6    function Deposit() public payable {
7      if (msg.value >= MinDeposit) {
8        balances[msg.sender] += msg.value;
9        TransferLog.AddMessage("Deposit");
10     }
11   }
12   function CashOut(uint _am) {
13     if (_am<=balances[msg.sender]) {
14       if (msg.sender.call.value(_am)()) {
15         balances[msg.sender] -= _am;
16         TransferLog.AddMessage("CashOut");
17       }
18     }
19   }
20 }
21 contract Log {
22   ...
23   function AddMessage(string _data) public {
24     LastMsg.Time = now;
25     LastMsg.Data = _data;
26     History.push(LastMsg);
27   }
28 }
```

**Figure 6.9:** An example of a straw man contract honeypot.

would transfer all the funds to the victim, making him or her believe that the contract is not a honeypot. We label this type of honeypot as *hidden transfer*.

**Straw Man Contract.** In Figure 6.9 we provide an example of a honeypot technique that we denote as *straw man contract*. At first sight, it seems that the contract's `CashOut` function is vulnerable to a reentrancy attack [60] (line 14). In order to be able to mount the reentrancy attack, the user is required to first call the `Deposit` function and transfer a minimum amount of ether. Eventually, the user calls the `CashOut` function, which performs a call to the contract address stored in `TransferLog`. As shown in the Figure 6.9, the contract called `Log` is supposed to act as a logger. However, the honeypot creator did not initialize the contract with an address containing the bytecode of the shown logger contract. Instead it has been initialized with another address pointing to a contract that implements the same interface, but throws an exception if the function `AddMessage` is called with the string "CashOut" and the caller is not the honeypot creator. Thus, the reentrancy attack performed by the user will always fail. Another alternative, is to use a `delegatecall` right before the transfer of the balance. Delegatecall allows a callee contract to modify the stack of the caller contract. Thus, the attacker would simply swap the address of the user contained on the stack with his or her

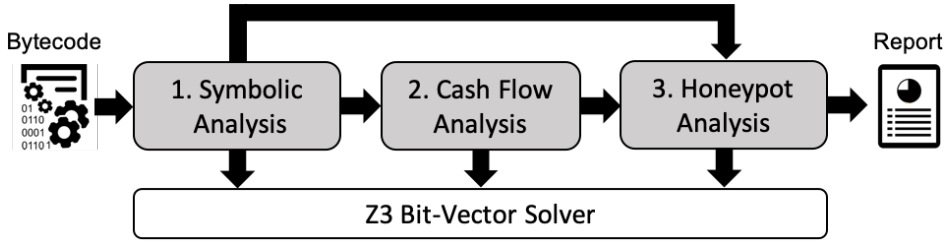



**Figure 6.10:** An overview of the analysis pipeline of HONEYBADGER. The shaded boxes represent the main components.

own address and when returning from the delegatecall, the balance would be transferred to the attacker instead of the user.

## 6.3 HONEYBADGER

In this section, we provide an overview on the design and implementation of HONEYBADGER[3].

### 6.3.1 Design Overview

Figure 6.10 depicts the overall architecture and analysis pipeline of HONEYBADGER. HONEYBADGER takes as input EVM bytecode and returns as output a detailed report regarding the different honeypot techniques it detected. HONEYBADGER consists of three main components: *symbolic analysis*, *cash flow analysis* and *honeypot analysis*. The symbolic analysis component constructs the Control-Flow Graph (CFG) and symbolically executes its different paths. The result of the symbolic analysis is afterwards propagated to the cash flow analysis component as well as the honeypot analysis component. The cash flow analysis component uses the result of the symbolic analysis to detect whether the contract is capable to receive as well as transfer funds. Finally, the honeypot analysis component aims at detecting the different honeypots techniques studied in this chapter using a combination of heuristics and the results of the symbolic analysis. Each of the three components uses the Z3 SMT solver [11] to check for the satisfiability of constraints.

### 6.3.2 Implementation

HONEYBADGER is implemented in Python, with roughly 4,000 lines of code. We briefly describe the implementation details of each main component below.

**Symbolic Analysis.** The symbolic analysis component starts by constructing a CFG from the bytecode, where every node in the CFG corresponds to a basic block and every edge

[3]Code is publicly available at: https://github.com/christoftorres/HoneyBadger

corresponds to a jump between individual basic blocks. A basic block is a sequence of instructions with no jumps going in or out of the middle of the block. The CFG captures all possible program paths that are required for symbolic execution. Symbolic execution represents the values of program variables as symbolic expressions. Each program path consists of a list of path conditions (a formula of symbolic expressions), that must be satisfied for execution to follow that path. We reused and modified the symbolic execution engine proposed by Luu et al. [51, 76]. The engine consists of an interpreter loop that receives a basic block and symbolically executes every single instruction within that block. The loop continues until all basic blocks of the CFG have been executed or a timeout is reached. Loops are terminated once they exceed a globally defined loop limit. The engine follows a depth first search approach when exploring branches and queries Z3 to determine their feasibility. A path is denoted as feasible if its path conditions are satisfiable. Otherwise, it is denoted as infeasible. Usually, symbolic execution tries to detect and ignore infeasible paths in order to improve their performance. However, our symbolic execution does not ignore infeasible paths, but executes them nevertheless, as they can be useful for detecting honeypots (see Section 6.3.2). The purpose of the symbolic analysis is to collect all kinds of information that might be useful for later analysis. This information includes a list of storage writes, a list of execution paths $P$, a list of infeasible as well as feasible basic blocks, a list of performed multiplications and additions, and a list of calls $C$. Calls are extracted through the opcodes CALL and DELEGATECALL, and either represent a function call, a contract call or a transfer of Ether. A call consists of the tuple $(c_r, c_v, c_f, c_a, c_t, c_g)$, where $c_r$ is the recipient, $c_v$ is the call value, $c_f$ is the called contract function, $c_a$ is the list of function arguments, $c_t$ is the type of call (i.e., CALL or DELEGATECALL) and $c_g$ is the available gas for the call.

**Cash Flow Analysis.** Given the definition in Section 6.2.1, a honeypot must be able to *receive* funds (e.g., the investment of a victim) and *transfer* funds (e.g., the loot of the attacker). The purpose of our *cash flow* analysis is to improve the performance of our tool, by safely discarding contracts that cannot receive or transfer funds.

*Receiving Funds.* There are multiple ways to receive funds besides direct transfers: as a recipient of a block reward, as a destination of a selfdestruct or through the call of a payable function. Receiving funds through a block reward or a selfdestruct makes little sense for a honeypot as this would not execute any harmful code. Also, the compiler adds a check during compilation time that reverts a transaction if a non-payable function receives a transaction value that is larger than zero. Based on these observations, we verify that a contract is able to receive funds, by first iterating over all possible execution paths contained in $P$ and checking whether there exists an execution path $p$ that does not terminate in a `REVERT`. Afterwards, we use Z3 to verify if the constraint $I_v > 0$ can be satisfied under the given path conditions of the execution path $p$. If $p$ satisfies the constraint, we know that funds can flow into the contract.

*Transferring Funds.* There are two different ways to transfer funds: either explicit via a *call* or implicit via a *selfdestruct*. We verify the former by iterating over all calls contained in $C$ and checking whether there exists a call $c$, where $c_v$ is either symbolic or $c_v > 0$. We verify the latter by iterating over all execution paths contained in $P$ and checking whether there exists an execution path $p$ that terminates in a `SELFDESTRUCT`. Finally, we know that funds can flow out of the contract, if we find at least one call $c$ or execution path $p$, that satisfies the aforementioned conditions.

**Honeypot Analysis.** The honeypot analysis consists of several sub-components. Each sub-component is responsible for the detection of a particular honeypot technique. Every honeypot technique is identified via heuristics. We describe the implementation of each sub-component below. The honeypot analysis can easily be extended to detect future honeypots by simply implementing new sub-components.

- **Balance Disorder.** Detecting a balance disorder is straightforward. We iterate over all calls contained in $C$ and report a balance disorder, if we find a call $c$ within an infeasible basic block, where $c_v = I_v + \sigma[I_a]_b$.
- **Inheritance Disorder.** Detecting an inheritance disorder at the bytecode level is rather difficult since bytecode does not include information about inheritance. Therefore, we leverage on implementation details that are specific to this honeypot technique: 1) there exists an $I_s$ that is written to a storage location which is never used inside a path condition, call or suicide; and 2) there exists a call $c$, whose path conditions contain a comparison between $I_s$ and a storage variable, whose storage location is different than the storage location identified in 1).
- **Skip Empty String Literal.** We start by iterating over all calls contained in $C$ and checking whether there exists a call $c$, where the number of arguments in $c_a$ is smaller than the number of arguments expected by $c_f$. We report a skip empty string literal, if we can find another call $c'$, that is called within function $c_f$ and where $c'_r$ originates from an argument in $c_a$.
- **Type Deduction Overflow.** We detect a type deduction overflow by iterating over all calls contained in $C$ and checking whether there exists a call $c$, where $c_v$ contains the result of a multiplication or an addition that has been truncated via an AND mask with the value `0xff`, which represents the maximum value of an 8-bit integer.
- **Uninitialized Struct.** We use a regular expression to extract the storage location of structs, whose first element is pointing at storage location zero within a basic block. Eventually, we report an uninitialized struct, if there exists a call $c \in C$, where either $c_v$ contains a value from a storage location of a struct or the path condition of $c$ depends on a storage location of a struct.

- **Hidden State Update.** We detect a hidden state update by iterating over all calls contained in $C$ and checking whether there exists a call $c$, whose path conditions depend on a storage value that can be modified via another function, without the transfer of funds.

- **Hidden Transfer.** We report a hidden transfer, if two consecutive calls $c$ and $c'$ exist along the same execution path $p$, where $c_r \in \sigma[I_a]_s \wedge c_v = \sigma[I_a]_b$ and $c'_r = I_s \wedge c'_v = \sigma[I_a]_b$.

- **Straw Man Contract.** First, we verify if two consecutive calls $c$ and $c'$ exist along the same execution path $p$, where $c_r \neq c'_r$. Finally, we report a straw man contract if one of the two cases is satisfied: 1) $c$ is executed after $c'$, where $c'_t = DELEGATECALL \wedge c_v = \sigma[I_a]_b \wedge c_r = I_s$; or 2) $c$ is executed before $c'$, where $c'_t = CALL \wedge I_s \in c'_a$.

## 6.4 Evaluation

In this section, we assess the correctness and effectiveness of HONEYBADGER. We aim to determine the reliability of our tool and measure the overall prevalence of honeypots currently deployed on the Ethereum blockchain.

**Dataset.** We downloaded the bytecode of 2,019,434 smart contracts, by scanning the first 6,500,000 blocks of the Ethereum blockchain. The timestamps of the collected contracts range from August 7, 2015 to October 12, 2018. Interestingly, a lot of contracts share the same bytecode. Out of the 2,019,434 contracts, only 151,935 are unique in terms of exact bytecode match. In other words, 92.48% of the contracts deployed on the Ethereum blockchain are duplicates. The most duplicated contract is replicated 387,914 times.

**Experimental Setup.** All experiments were conducted on our high-performance computing cluster using 10 nodes with 960 GB of memory, where every node has 2 Intel Xeon L5640 CPUs with 12 cores each and clocked at 2,26 GHz, running 64-bit Debian Jessie 8.10. We used version 1.8.16 of Geth's EVM as our disassembler and Solidity version 0.4.25 as our source-code-to-bytecode compiler. As our constraint solver we used Z3 version 4.7.1. We set a timeout of 1 second per Z3 request for the symbolic execution. The symbolic execution's global timeout was set to 30 minutes per contract. The loop limit, depth limit (for DFS) and gas limit for the symbolic execution were set to 10, 50 and 4 million, respectively.

### 6.4.1 Results

We run HONEYBADGER on our set of 151,935 unique smart contracts. Our tool took an average of 142 seconds to analyse a contract, with a median of 31 seconds and a mode of less than 1 second. Moreover, for 98% of the cases (149,603 contracts) our tool was able

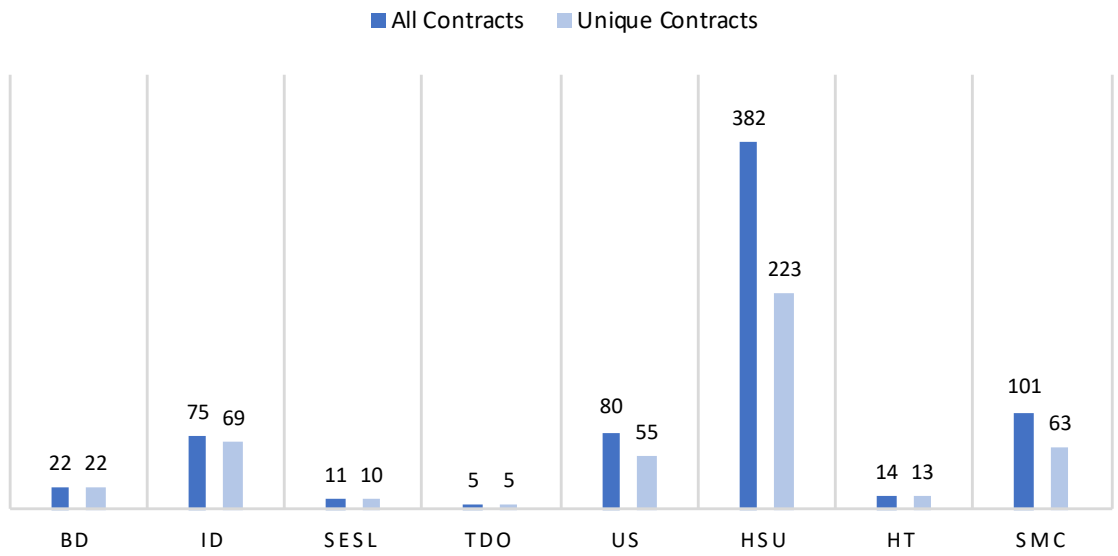



**Figure 6.11:** Number of detected honeypots per technique.

to finish its analysis within the given time limit of 30 minutes. The number of explored paths ranges from 1 to 8,037, with an average of 179 paths per contract and a median of 105 paths. Finally, during our experiments, HONEYBADGER achieved a code coverage of about 91% on average. Out of the 151,935 analyzed contracts, 48,487 have been flagged as cash flow contracts. In other words, only 32% of the analysed contracts are capable of receiving as well as sending funds. Figure 6.11 depicts for each honeypot technique the number of contracts that have been flagged by HONEYBADGER. Our tool detected a total of 460 unique honeypots. It is worth mentioning that 24 out of the 460 honeypots were part of our initial dataset (see Table 6.1) and that our tool thus managed to find 436 new honeypots. Moreover, as mentioned earlier, many contracts share the same bytecode. Thus, after correlating the results with the bytecode of the 2 million contracts currently deployed on the blockchain, a total of 690 contracts were identified as honeypots[4]. Our tool therefore discovered a total of 22 balance disorders (BD), 75 inheritance disorders (ID), 11 skip empty string literal (SESL), 5 type deduction overflows (TDO), 80 uninitialized structs (US), 382 hidden state updates (HSU), 14 hidden transfers (HT) and finally 101 straw man contracts (SMC). While many contracts were found to be HSU, SMC and US honeypots, only a small number were found to be TDO honeypots.

### 6.4.2 Validation

In order to confirm the correctness of HONEYBADGER, we performed a manual inspection of the source code of the contracts that have been flagged as honeypots. We were able to collect through Etherscan the source code for 323 (70%) of the flagged contracts. We verified the flagged contracts by manually scanning the source code for characteristics of

[4]The latest honeypots can be found at: https://honeybadger.uni.lu/

the detected honeypot technique. For example, in case a contract has been flagged as a balance disorder, we checked whether the source code contains a function that transfers the contract's balance to the caller if and only if the value sent to the function is greater than or equal to the contract's balance.

**Table 6.3:** Number of true positives (TP), false positives (FP) and precision $p$ (in %) per detected honeypot technique for contracts with source code.

| | Balance Disorder | Inheritance Disorder | Skip Empty String Literal | Type Deduction Overflow | Uninitialised Struct | Hidden State Update | Hidden Transfer | Straw Man Contract |
|---|---|---|---|---|---|---|---|---|
| TP | 20 | 41 | 9 | 4 | 32 | 134 | 12 | 30 |
| FP | 0 | 7 | 0 | 0 | 0 | 30 | 0 | 4 |
| $p$ | 100 | 85 | 100 | 100 | 100 | 82 | 100 | 88 |

Table 6.3 summarises our manual verification in terms of true positives (TP), false positives (FP) and precision $p$, where $p$ is computed as $p = TP/(TP + FP)$. A true positive means that the contract is indeed a honeypot with respect to the reported technique and a false positive means that the contract is *not* a honeypot with respect to the reported technique. Overall our tool shows a very high precision and a very low false positive rate. Our tool achieves a false positive rate of 0% for 5 out of the 8 analyzed honeypot techniques. For the remaining 3 techniques, our tool achieves a decent false positive rate, where the highest false positive rate is roughly 18% for the detection of hidden state updates, followed by 15% false positive rate for the detection of inheritance disorder and finally 12% false positive rate for the detection of straw man contracts.

## 6.5 Analysis

In this section, we analyze the true positives obtained in Section 6.4, in order to acquire insights on the *effectiveness*, *liveness*, *behavior*, *diversity* and *profitability* of honeypots.

### 6.5.1 Methodology

We crawled all the transactions of the 282 true positives using Etherchain's[5] API, in order to collect various information about the honeypots, such as the amount of spent and re-

[5]https ://www.etherchain.org/

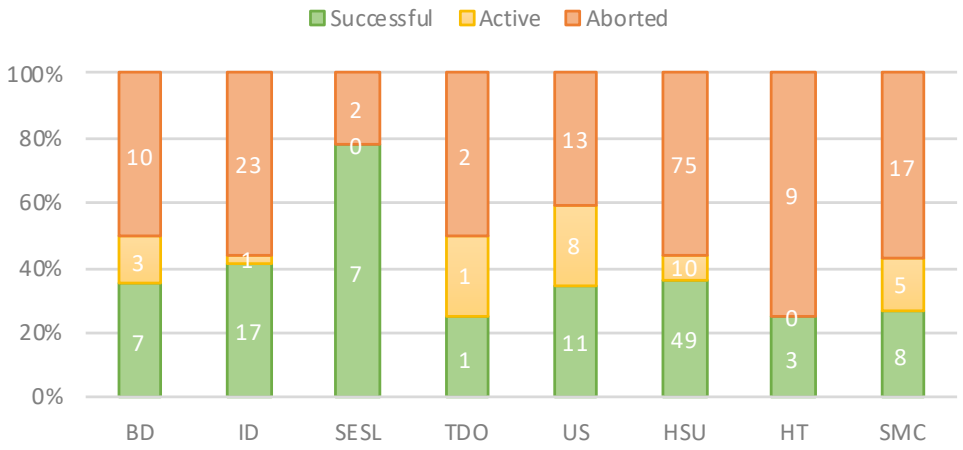



**Figure 6.12:** Number of successful, active and aborted honeypots per honeypot technique.

ceived ether per address, the deployment date and the balance. Afterwards, we used simple heuristics to label every address as either an *attacker* or a *victim*. An address is labeled as an attacker if it either: 1) created the honeypot; 2) was the first address to send ether to the honeypot; or 3) received more ether than it actually spent on the honeypot. An address is labeled as a victim if it has not been labeled as an attacker and if it received less ether than it actually spent on the honeypot. Finally, using this information we were able to tell if a honeypot, was either *successful*, *aborted* or still *active*. A honeypot is marked as successful if a victim has been detected, as aborted if the balance is zero and no victim has been detected or as active if the balance is larger than zero and no victim has been detected.

### 6.5.2 Results

**Effectiveness.** Figure 6.12 shows the number of successful, aborted and active honeypots per honeypot technique. Our results show that *skip empty string literal* is the most effective honeypot technique with roughly 78% success rate, whereas *hidden transfer* is the least effective technique with only 33% success rate. The overall success rate of honeypots seems rather low with roughly 37%, whereas the overall abortion rate seems quite high with about 54%. At the time of writing, only 10% of the analyzed honeypots are still active. Figure 6.13 illustrates the number of monthly deployed honeypots per honeypot technique. The very first honeypot technique that has been deployed was a hidden state update in January 2017. February 2018 has been the peak in terms of honeypots being deployed, with a total of 66. The highest number of monthly honeypots that have been deployed per technique are hidden state updates with a total of 36 in June 2018. 7 honeypots have been deployed on average per month. In our analysis, the quickest first attempt of exploitation happened just 7 minutes and 37 seconds after a honeypot had been deployed, whereas the longest happened not until 142 days after deployment. A honeypot takes an average

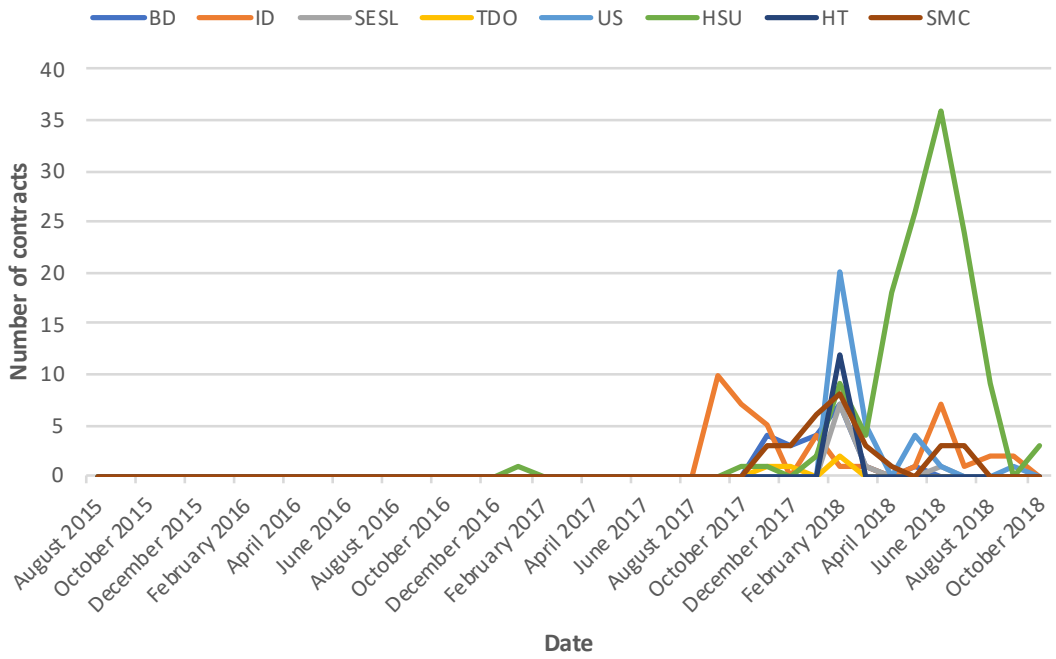



**Figure 6.13:** Number of monthly deployed honeypots per honeypot technique.

of 9 days and a median of 16 hours before it gets exploited. Interestingly, most honeypots (roughly 55%) are exploited during the first 24 hours after being deployed.

**Liveness.** We define the lifespan of a honeypot as the period of time between the deployment of a honeypot and the moment when a honeypot was aborted. We found that the shortest lifespan of a honeypot was 5 minutes and 25 seconds and the longest lifespan was about 322 days. The average lifespan of a honeypot is roughly 28 days, whereas the median is roughly 3 days. However, in around 32% of the cases the lifespan of a honeypot is only 1 day. We also analyzed how long an attacker keeps the funds inside a honeypot, by measuring the period of time between the first attempt of exploitation by a victim and the withdrawal of all the funds by the attacker. The shortest period was just 4 minutes and 28 seconds after a victim fell for the honeypot. The longest period was roughly 100 days. On average attackers withdraw all their funds within 7 days after a victim fell for the honeypot. However, in most cases the attackers keep the funds in the honeypot for a maximum of 1 day. Interestingly, only 37 out of 282 honeypots got destroyed, where destroyed means that the attacker called a function within the honeypot that calls the SELFDESTRUCT opcode. In other words, 171 honeypots are in some kind of “zombie” state, where they are still alive (i.e., not destroyed), but not active (i.e., their balance is zero). Analyzing the 37 destroyed honeypots, we found that 19 got destroyed after being successful and 18 after never having been successful.

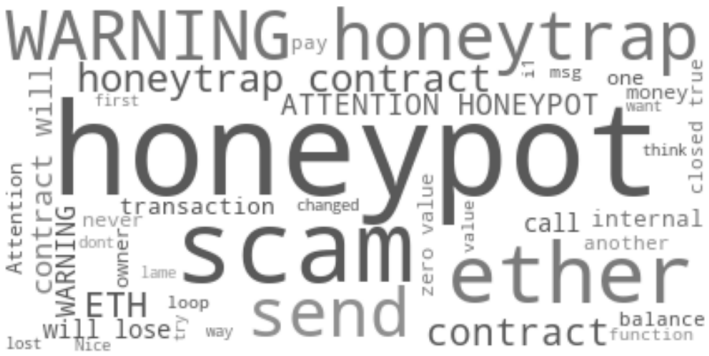



**Figure 6.14:** A word cloud generated from the comments on Etherscan.

**Behavior.** Our methodology classified a total of 240 addresses as victims. In 71% of the cases a honeypot managed to trap only one victim. In one case though, 97 victims have been trapped by just a single honeypot. Interestingly, 8 out of the 240 addresses fell for more than one honeypot, where one address even became a victim to four different honeypots. We also found that 53 attackers deployed at least two honeypots, whereas one attacker deployed eight different honeypots. It is worth noting that 42 of the 53 attackers simply deployed copies of one particular honeypot type, whereas the remaining 11 deployed honeypots of varying types. 87 out of the 282 detected and manually confirmed honeypots (about 31%) contained comments on Etherscan. We manually analyzed these comments and found that the majority of the comments were indeed warnings stating that the contract might be a honeypot. Moreover, Figure 6.14 shows that the term “honeypot” is the most prevalent term used by the community to describe this type of smart contracts. Surprisingly, 20 out of the 87 commented honeypots were successful. 16 were successful before a comment had been placed and 4 have been successful even after a comment had been placed. Interestingly, 21 honeypots aborted after a comment was placed. The quickest abort was performed just 33 minutes and 57 seconds after the comment, whereas the longest abort was performed 37 days after the comment. Finally, attackers took an average of 6 days and a median of 22 hours to abort their honeypot after a user had placed a comment.

**Diversity.** We used the normalized Levenshtein distance [9] to measure the similarity of the bytecode between the individual instances of a particular honeypot technique. Table 6.4 outlines the similarity in terms of minimum, maximum, mean and mode per honeypot technique. We observe that for almost every technique, except TDO, the bytecode similarity varies tremendously. For example, in case of hidden state update honeypots, we measure a minimum similarity of 11% and a maximum similarity of 98%. This indicates that even though two honeypots share the same technique, their bytecode might still be very diverse.

**Profitability.** Table 6.5 lists the profitability per honeypot technique. The profitability is computed as `received amount` - (`spent amount` + `transaction fees`). No values are pro-

**Table 6.4:** Bytecode similarity (in %) per honeypot technique.

| | BD | ID | SESL | TDO | US | HSU | HT | SMC |
|---|---|---|---|---|---|---|---|---|
| Min. | 27 | 14 | 22 | 88 | 25 | 11 | 28 | 26 |
| Max. | 97 | 96 | 98 | 95 | 98 | 98 | 98 | 98 |
| Mean | 50 | 40 | 47 | 90 | 52 | 49 | 71 | 53 |
| Mode | 35 | 35 | 28 | 89 | 45 | 36 | 95 | 49 |

vided for TDO, because for the single true positive that we analyzed, the transaction fees spent by the attacker were higher than the amount that the attacker gained from the victim. The smallest and largest profit were made using a hidden state update honeypot, with 0.00002 ether being the smallest and 11.96 ether being the largest. The most profitable honeypots are straw man contract honeypots, with an average value of 1.76 ether, whereas the least profitable honeypots are uninitialized struct honeypots, with an average value of 0.46 ether. A total profit of 257.25 ether has been made through honeypots, of which 171.22 ether were made through hidden state update honeypots. Note that, the exchange rate of cryptocurrencies is very volatile and thus their value in USD may vary greatly on a day-to-day basis. For example, although 11.96 ether is the largest profit made in ether, its actual value in USD was only 500 at the point of withdrawal. We found that the largest profit in terms of USD was actually a honeypot with 3.10987 ether, as it was worth 2,609 USD at the time of withdrawal. Applying this method across the 282 honeypots, results in a total profit of 90,118 USD.

## 6.6 Discussion

In this section, we summarize the key insights gained through our analysis and we discuss the ethical considerations as well as the challenges and limitations of our work.

**Table 6.5:** Statistics on the profitability of each honeypot technique in ether.

| | Min. | Max. | Mean | Mode | Median | Sum |
|---|---|---|---|---|---|---|
| BD | 0.01 | 1.13 | 0.5 | 0.11 | 0.11 | 3.5 |
| ID | 0.004 | 6.41 | 1.06 | 0.1 | 0.33 | 17.02 |
| SESL | 0.584 | 4.24 | 1.59 | 1.0 | 1.23 | 9.57 |
| TDO | - | - | - | - | - | - |
| US | 0.009 | 1.1 | 0.46 | 0.1 | 0.38 | 6.44 |
| HSU | 0.00002 | 11.96 | 1.44 | 0.1 | 1.02 | 171.22 |
| HT | 1.009 | 1.1 | 1.05 | 1.0 | 1.05 | 2.11 |
| SMC | 0.399 | 4.94 | 1.76 | 2.0 | 1.99 | 47.39 |
| Overall | 0.00002 | 11.96 | 1.35 | 1.0 | 1.01 | 257.25 |

### 6.6.1 Honeypot Insights

Although honeypots are capable of trapping multiple users, we have found that most honeypots managed to take the funds of only one victim. This indicates that users potentially look at the transactions of other users before they submit theirs. Moreover, the low success rate of honeypots with comments, suggests that users also check the comments on Etherscan before submitting any funds. We also found that the bytecode of honeypots can be vastly different even if using the same honeypot technique. This suggests that the usage of signature-based detection methods would be rather ineffective. HONEYBADGER is capable of recognizing a variety of implementations, as it specifically targets the functional characteristics of each honeypot technique. More than half of the honeypots were successful within the first 24 hours. This suggests that honeypots become less effective the older they become. This is interesting, as it means that users seem to target rather recently deployed honeypots than older ones. We also note that most honeypot creators withdraw their loot within 24 hours or abort their honeypots if they are not successful within the first 24 hours. We therefore conclude that honeypots have in general a short lifespan and only a small fraction remain active for a period longer than one day.

### 6.6.2 Challenges and Limitations

The amount of smart contracts with source code available is rather small. In 2019, only 50,000 contracts with source code are available on Etherscan. This highlights the necessity of being able to detect honeypots at the bytecode level. Unfortunately, this turns out to be extremely challenging when detecting certain honeypot techniques. For example, while detecting inheritance disorder at the source code level is rather trivial, detecting it at the bytecode level is rather difficult since all information about the inheritance is lost during compilation and not available anymore at the bytecode level. The fact that certain information is only available at the source code level and not at the bytecode level, obliges us to make use of other less precise information that is available in the bytecode in order to detect honeypot techniques such as inheritance disorder. However, as Section 6.4 shows, this approach reduces the precision of our detection and introduces some false positives. Finally, another limitation of our tool is that it is currently limited to the detection of the eight honeypot techniques described in this chapter. Other honeypot techniques are not detected. Nevertheless, we designed HONEYBADGER with modularity in mind, such that one can easily extend the honeypot analysis with new heuristics to detect more honeypot techniques.

### 6.6.3 Ethical Considerations

In general, honeypots have two participants, the creator of the honeypot, and the user whose funds are trapped by the honeypot. The ethical intentions of both participants are not always clear. For instance, a honeypot creator might deploy a honeypot with the intention to scam

users and make profit. In this case we clearly have a malicious intention. However, one could also argue that a honeypot creator is just attempting to punish users that behave maliciously. Similarly, the intentions of a honeypot user can either be malicious or benign. For example, if a user tries to intentionally exploit a reentrancy vulnerability, then he or she needs to be knowledgeable and mischievous enough to prepare and attempt the attack, and thus clearly showing malicious behavior. However, if we take the example of an uninitialized struct honeypot that is disguised as a simple lottery, then we might have the case of a benign user who loses his funds under the assumption that he or she is participating in a fair lottery. Thus, neither honeypot creators nor users can always be clearly classified as either malicious or benign, this depends on the case at hand. Nevertheless, we are aware that our methodology may serve malicious attackers to protect themselves from other malicious attackers. However, with HONEYBADGER, we hope to raise the awareness of honeypots and save benign users from potential financial losses.

## 6.7 Related Work

Honeypots are a new type of fraud that combine security issues with scams. They either rely on the blockchain itself or on related services such as Etherscan. With growing interest within the blockchain community, they have been discussed online [130, 131, 132] and collected within public user repositories [118, 139]. Frauds and security issues are nothing new within the blockchain ecosystem. Blockchains have been used for money laundering [23] and been the target of several scams [31], including mining scams, wallet scams and Ponzi schemes, which are further discussed in [86, 138]. In particular, smart contracts have been shown to contain security issues [60]. Although not performed directly on the blockchain, exchanges have also been the target of fraud [22].

Several different methods have been proposed to discover fraud as well as security issues. Manual analysis is performed on publicly available source code to detect Ponzi schemes [62]. [140] introduces ERAYS, a tool that aims to produce easy to analyze pseudocode from bytecode where the source code is not available. However, manual analysis is particularly laborious, especially considering the number of contracts on the blockchain. Machine learning has been used to detect Ponzi schemes [92] and to find vulnerabilities [134]. The latter relies on [122] to obtain a ground truth of vulnerable smart contracts for training their model. Fuzzing techniques have been employed to detect security vulnerabilities in smart contracts [110] and in combination with symbolic execution to discover issues related to the ordering of events or function calls [115]. However, fuzzing often fails to create inputs to enter specific execution paths and therefore might ignore them [137]. Static analysis has been used to find security [90, 136, 135] and gas-focused [103] vulnerabilities in smart contracts. [90] requires manual interaction, while [136] requires both the definition of violation and compliance patterns. [135] requires Solidity code and therefore cannot be used to an-

alyze the large majority of the smart contracts deployed on the Ethereum blockchain. [103] considers gas-related issues which is not necessary for the purpose of this work. In order to use formal verification, smart contracts can, to some extent, be translated from source code or bytecode into F* [34, 104] where the verification can more easily be performed. Other work operates on high-level source code available for Ethereum or Hyperledger [112]. [70, 71] propose a formal definition of the EVM, that is extended in [85] towards more automated smart contract verification and the consideration of gas. Formal verification often requires (incomplete) translations into other languages or manual user interaction (e.g., [129]). Both of these reasons make formal verification unsuitable to perform a large analysis of smart contracts, as it is required in this work.

Symbolic execution has been used on smart contracts to detect common [87, 120, 51, 102] vulnerabilities. This technique also allows to find specific kinds of misbehaving contracts [122]. It can further provide values that can serve to generate automated exploits that trigger vulnerabilities [116]. The same technique is used in this chapter. Symbolic execution has the advantage of being capable to reason about all possible execution paths and states in a smart contract. This allows for the implementation of precise heuristics while achieving a low false positive rate. Another advantage is that symbolic execution can be applied directly to bytecode, thus making it well suited for our purpose of analyzing more than 2 million smart contracts for which source code is largely not available. The disadvantage is the large number of possible paths that need to be analyzed. However, in the case of smart contracts this is mostly not an issue, as most are not very complex and rather short. Moreover, smart contract bytecode cannot grow arbitrarily large due to the gas limit enforced by the Ethereum blockchain.

## 6.8 Conclusion

We investigated an emerging new type of fraud in Ethereum: *honeypots*. We presented a taxonomy of honeypot techniques and introduced a methodology that uses symbolic execution together with well-defined heuristics to automatically detect honeypots. We showed that HONEYBADGER can effectively detect honeypots in the wild with a very low false positive rate. In a large-scale analysis of 151,935 unique Ethereum smart contracts, HONEYBADGER identified 460 unique honeypots. Moreover, an analysis on the transactions performed by a subset of 282 honeypots, revealed that 240 users already became victims of honeypots and that attackers already made more than 90,000 USD profit with honeypots. It is worth noting that these numbers only provide a lower bound and thus might only reflect the tip of the iceberg. Nonetheless, tools such as HONEYBADGER may already help users in detecting honeypots before they can cause any harm.

# 7 | Frontrunner Jones

## *Measuring Frontrunning Attacks on Smart Contracts*

*In this chapter, we measure the prevalence of frontrunning attacks on smart contracts. Ethereum prospered the inception of a plethora of smart contract applications, ranging from gambling games to decentralized finance. However, Ethereum is also considered a highly adversarial environment, where vulnerable smart contracts will eventually be exploited. Recently, Ethereum's pool of pending transaction has become a far more aggressive environment. In the hope of making some profit, attackers continuously monitor the transaction pool and try to frontrun their victims' transactions by either displacing or suppressing them, or by strategically inserting their own transactions. This chapter aims to shed some light into what is known as a* dark forest *and uncover these predators' actions. We present a methodology to efficiently measure the three types of frontrunning:* displacement*,* insertion*, and* suppression*. We perform a large-scale analysis on more than 11M blocks and identify almost 200K attacks with an accumulated profit of 18.41M USD for the attackers, providing evidence that frontrunning is both, lucrative and a prevalent issue.*

## 7.1 Introduction

The concept of frontrunning is not new. In financial markets, brokers act as intermediaries between clients and the market, and thus brokers have an advantage in terms of insider knowledge about potential future buy/sell orders which can impact the market. In this context, frontrunning is executed by prioritizing a broker's trading actions before executing the client's orders such that the trader pockets a profit. Frontrunning is illegal in regulated financial markets. However, the recent revolution enabled by decentralized finance (DeFi), where smart contracts and miners replace intermediaries (brokers) is both a blessing and a curse. Removing trusted intermediaries can streamline finance and substantially lower adjacent costs, but misaligned incentives for miners leads to generalized frontrunning, in which market participants behave exactly like unethical brokers used to in the "old" financial world. Unfortunately, this is already happening at a large scale.

Our work is among the first comprehensive surveys on the extent and impact of this phenomenon. Already in 2017, the Bancor ICO [82] was susceptible to such an attack – among

other vulnerabilities – but no real attack was observed in the wild. Some concrete frontrunning attacks on the Ethereum blockchain were brought to knowledge by two independently reported attacks and their mitigation approaches to the informed audience. In the first report [227], the researchers tried to recover some liquidity tokens by calling a specific function in a smart contract. Since this function was callable by everyone, the authors – who also compared the pending transactions in the transaction pool to a *dark forest* full of predators – assumed that their function call could be observed and frontrunned by bots observing the submitted transactions in the transaction pool. Even though they tried to obfuscate their efforts, their approach failed in the end, and they became a victim of a frontrunning bot. A few months later, a second group of researchers [232] reported a successful recovery using lessons learned from the previously reported incident [227]. The success was due to them mining their transactions privately without broadcasting them to the rest of the network. The researchers used a new functionality provided by SparkPool called the Taichi Network [228]. In this way, the transactions were not available to frontrunning bots but relied entirely on having a reliable and honest mining pool. However, this approach enables centralization and requires users to entrust their transactions to SparkPool. Similar to how honeypots gather intelligence by luring attackers to compromise apparently vulnerable hosts [5], a recent experiment [229] detailed the interactions with two bots and reported relevant assessment on their nature and origin. Surprisingly, the frontrunning bots do not rely on advanced software development techniques or complex instructions, and code examples on developing such bots are readily available [230, 231].

There are several ways to perform frontrunning attacks. The first survey defining a taxonomy of frontrunning attacks [151] identified three different variants on how these can be performed. To understand these approaches – *displacement*, *insertion*, and *suppression* – a short refresh on gas and transaction fees in Ethereum is given. Transactions, submitted to the Ethereum network, send money and data to smart contract addresses or account addresses. Transactions are confirmed by miners who get paid via a fee that the sender of the transaction pays. This payment is also responsible for the speed/priority miners include a transaction in a mined block. Miners have an inherent incentive to include high paying transactions and prioritize them. As such, nodes observing the unconfirmed transactions can frontrun by just sending transactions with higher payoffs for miners [150]. The common feature of all three attack types is that by frontrunning a transaction, the initial transaction's expected outcome is changed. In the case of the first attack (displacement), the outcome of a victim's original transaction is irrelevant. The second attack type (insertion) manipulates the victim's transaction environment, thereby leading to an arbitrage opportunity for the attacker. Finally, the third attack (suppression) delays the execution of a victim's original transaction. Although previous chapters [150, 151] have identified decentralized applications which are victims of frontrunning attacks, no scientific study has analyzed the occurrence of these three attacks in the wild on a large scale. The impact of this structural design failure of the

Ethereum blockchain is far-reaching. Many decentralized exchanges, implementing token-based market places have passed the 1B USD volume [233] and are prone to the same frontrunning attack vectors because the Ethereum blockchain is used as a significant building block. Frontrunning is not going to disappear any time soon, and the future looks rather grim. We do not expect to have mitigation against frontrunning in the short-term. Miners do profit from the fees and thus will always prioritize high-yield transactions. Moreover, the trust mechanism in Ethereum is built on the total observability of the confirmed/unconfirmed transactions and is thus by design prone to frontrunning.

This chapter sheds light into the long-term history of frontrunning on the Ethereum blockchain and provides the first large scale data-driven investigation of this type of attack vector. We investigate the real profits made by attackers, differentiated by the specific attack type and propose the first methodology to detect them efficiently. In summary, this chapter makes the following contributions:

**Contributions**

- We propose a methodology that is efficient enough to detect *displacement*, *insertion*, and *suppression* attacks on Ethereum's past transaction history.
- We run an extensive measurement study and analyze frontrunning attacks on Ethereum for the past five years.
- We identify a total of 199,725 attacks, 1,580 attacker accounts, 526 bots, and over 18.41M USD profit.
- We demonstrate that the identified attacker accounts and bots can be grouped to 137 unique attacker clusters.
- We discuss frontrunning implications and find that miners made a profit of 300K USD due to frontrunners.

## 7.2 Frontrunning Attacks

This section defines our attacker model and introduces the reader to three different types of frontrunning attacks.

### 7.2.1 Attacker Model

Miners, as well as non-miners, can mount frontrunning attacks. Miners are not required to pay a higher gas price to manipulate the order of transactions as they have full control over how transactions are included. Non-miners, on the other hand, are required to pay a higher gas price in order to frontrun transactions of other non-miners. Our attacker model assumes an attacker $A$ that is a financially rational non-miner with the capability to monitor

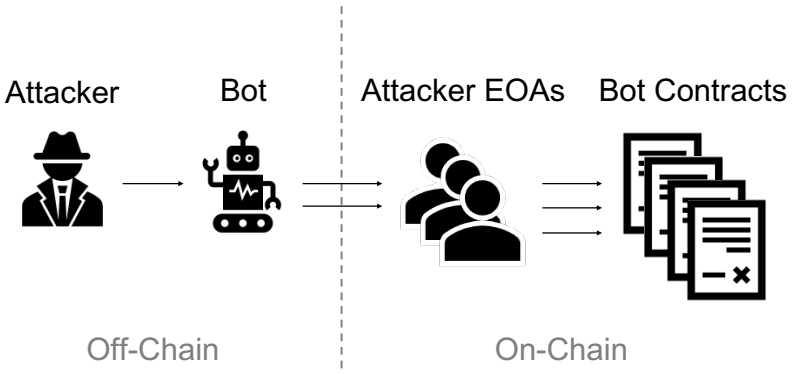



**Figure 7.1:** Attacker model with on-chain and off-chain parts.

the transaction pool for incoming transactions. The attacker $A$ needs to process the transactions in the pool, find a victim $V$ among those transactions and create a given amount of attack transactions $T_{A_i}$ before the victim's transaction $T_V$ is mined. Usually, $A$ would not be able to react fast enough to perform all these tasks manually. Hence, we assume that the attacker $A$ has at least one computer program $Bot_A$ that automatically performs these tasks. However, $Bot_A$ must be an off-chain program, because contracts cannot react on its own when transactions are added to the pool. Nevertheless, $Bot_A$ needs at least one or more EOAs to act as senders of any attack transaction $T_A$. Using multiple EOAs helps attackers obscure their frontrunning activities, similar to money laundering layering schemes. We refer to these EOAs owned by $A$ as attacker accounts $EOA_{A_j}$ and to the EOA owned by $V$ as victim account $EOA_V$. We assume that attacker $A$ owns a sufficiently large balance across all its attacker accounts $EOA_{A_j}$ from which it can send frontrunning transactions. However, attacker $A$ can also employ smart contracts to hold part of the attack logic. We refer to these smart contracts as bot contracts $BC_{A_k}$, which are called by the attacker accounts $EOA_{A_j}$. Figure 7.1 provides an overview of our final attacker model.

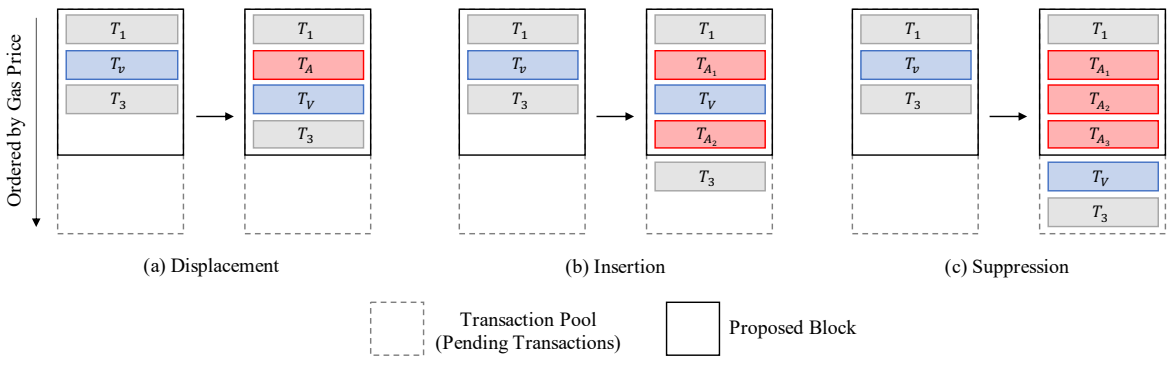



**Figure 7.2:** Illustrative examples of the three frontrunning attack types.

### 7.2.2 Frontrunning Taxonomy

We describe in the following the taxonomy of frontrunning attacks presented by Eskandari et al. [151].

**Displacement.** In a displacement attack an attacker $A$ observes a profitable transaction $T_V$ from a victim $V$ and decides to broadcast its own transaction $T_A$ to the network, where $T_A$ has a higher gas price than $T_V$ such that miners will include $T_A$ before $T_V$ (see Figure 7.2 a). Note that the attacker does not require the victim's transaction to execute successfully within a displacement attack. For example, imagine a smart contract that awards a user with a prize if they can guess the preimage of a hash. An attacker can wait for a user to find the solution and to submit it to the network. Once observed, the attacker then copies the user's solution and performs a displacement attack. The attacker's transaction will then be mined first, thereby winning the prize, and the user's transaction will be mined last, possibly failing.

**Insertion.** In an insertion attack an attacker $A$ observes a profitable transaction $T_V$ from a victim $V$ and decides to broadcast its own two transactions $T_{A_1}$ and $T_{A_2}$ to the network, where $T_{A_1}$ has a higher gas price than $T_V$ and $T_{A_2}$ has a lower gas price than $T_V$, such that miners will include $T_{A_1}$ before $T_V$ and $T_{A_2}$ after $T_V$ (see Figure 7.2 b). This type of attack is also sometimes called a *sandwich attack*. In this type of attack, the transaction $T_V$ must execute successfully as $T_{A_2}$ depends on the execution of $T_V$. A well-known example of insertion attacks is arbitraging on decentralized exchanges, where an attacker observes a large trade, also known as a whale, sends a buy transaction before the trade, and a sell transaction after the trade.

**Suppression.** In a suppression attack, an attacker $A$ observes a transaction $T_V$ from a victim $V$ and decides to broadcast its transactions to the network, which have a higher gas price than $T_V$ such that miners will include $A$'s transaction before $T_V$ (see Figure 7.2 c). The goal of $A$ is to suppress transaction $T_V$, by filling up the block with its transactions such that transaction $T_V$ cannot be included anymore in the next block. This type of attack is also called *block stuffing*. Every block in Ethereum has a so-called *block gas limit*. The consumed gas of all transactions included in a block cannot exceed this limit. $A$'s transactions try to consume as much gas as possible to reach this limit such that no other transactions can be included. This type of attack is often used against lotteries where the last purchaser of a ticket wins if no one else purchases a ticket during a specific time window. Attackers can then purchase a ticket and mount a suppression attack for several blocks to prevent other users from purchasing a ticket themselves. Keep in mind that this type of frontrunning attack is expensive.

## 7.3 Measuring Frontrunning Attacks

This section provides an overview of our methodology's design and implementation details to detect frontrunning attacks in the wild.

### 7.3.1 Identifying Attackers

As defined in Section 7.2.1, an attacker $A$ employs one or more off-chain programs to perform its attacks. However, because we have no means to distinguish between the different software agents an attacker $A$ could have, for this study, we consider all of them as part of the same multi-agent system $Bot_A$. Additionally, we cannot recognize the true nature of $A$ or how $Bot_A$ is implemented. Instead, we would like to build a cluster with the $n$ different attacker accounts $EOA_{A_1}, \ldots, EOA_{A_n}$ and the $m$ different bot contracts $BC_{A_1}, \ldots, BC_{A_m}$ to form an identity of $A$. Consequently, in each of the following experiments, we use our detection system's results to build a graph. Each node is either an attacker account or a bot contract. We make the following two assumptions:

**Assumption 1:** Attackers only use their own bot contracts. Hence, when an attacker account sends a transaction to a bot contract, we suspect that both entities belong to the same attacker. Note that one attacker account can send transactions to multiple bot contracts, and bot contracts can receive transactions from multiple attacker accounts.

**Assumption 2:** Attackers develop their own bot contracts, and they do not publish the source code of their bot contracts as they do not want to share their secrets with competitors. Hence, when the bytecode of two bot contracts is exactly the same, we suspect that they belong to the same attacker.

With these assumptions in mind, we create edges between attacker accounts and bot contracts that share at least one attack transaction, and between bots that share the same bytecode. Using the resulting graph, we compute all the connected components. Hence, we interpret each of these connected components as a single attacker cluster.

### 7.3.2 Detecting Displacement

Attackers typically perform displacement attacks by observing profitable pending transactions via the transaction pool and by copying these profitable transactions' input to create and submit their own profitable transactions. While attackers are not required to use a bot contract to mount displacement attacks, using a smart contract allows them to limit their loss as they can abort the execution in case of an unexpected event. However, detecting displacement attacks that directly interact with the contract that is susceptible to displacement is tremendously hard as there is no possible way to distinguish between an attacker and a

benign user that just happened to send a transaction to the susceptible contract. Our detection is therefore limited towards finding attackers that perform displacement attacks using bot contracts. The general idea is to detect displacement by checking for every transaction $T$ if there exists a subsequent transaction $T'$ with a gas price lower than $T$ and a transaction index higher than $T$, where the input of $T'$ is contained inside the input of $T$. However, detecting displacement in the wild can become quite challenging due to a large number of possible combinations. A naive approach would be to obtain a list of every internal and external transaction per contract and then compare every transaction to every subsequent transaction. However, given that a single contract can have easily thousands of transactions, this approach would quickly result in a combinatorial explosion. Moreover, obtaining internal transactions requires re-executing all past transactions which results in a significant amount of time given that the Ethereum blockchain currently has more than 1 billion transactions. Our goal is therefore to focus only on external transactions and follow a more efficient approach that might sacrifice completeness but preserve soundness. We begin by splitting the range of blocks that are to be analyzed into windows of 100 blocks and slide them with an offset of 20 blocks. This approach has the advantage that each window can be analyzed in parallel. Inside each window, we iterate block by block, transaction by transaction, and split the input bytes of each transaction into $n$-grams of 4 bytes with an offset of 1 byte and check whether at least 95% of the $n$-grams match with $n$-grams of previous transaction inputs. Since we focus on detecting displacement attacks performed via bot contracts, we cannot use 100% matching, because the victim's external transaction will be encapsulated inside the attacker's external transaction along with some command-and-control data. Each window has its own Bloom filter that memorizes previously observed $n$-grams. A Bloom filter is a probabilistic data structure that can quickly tell if an element has already been observed before or if it definitely has not been observed before. Thus, Bloom filters may yield false positives, but no false negatives. The idea is first to use a Bloom filter to perform a quick probabilistic search and only perform an exhaustive linear search if the filter finds that at least 95% of a transaction's $n$-grams are contained in the filter. Our Bloom filters can hold up to $n$ = 1M elements with a false positive rate $p$ = 1%, which according to Bloom [1], requires having $k$ = 6 different hash functions:

$$m = -\frac{n \ln p}{(\ln 2)^2} \tag{7.1}$$

$$k = \frac{m}{n} \ln 2 \tag{7.2}$$

We bootstrapped our 6 hash functions using the Murmur3 hash function as a basis. The result of each hash function is an integer that acts as an index on the Bloom filter's bit array. The bit array is initialized at the beginning with zeros, and a value of one is set for each index returned by a hash function (see Figure 7.3). An $n$-gram is considered to be contained in

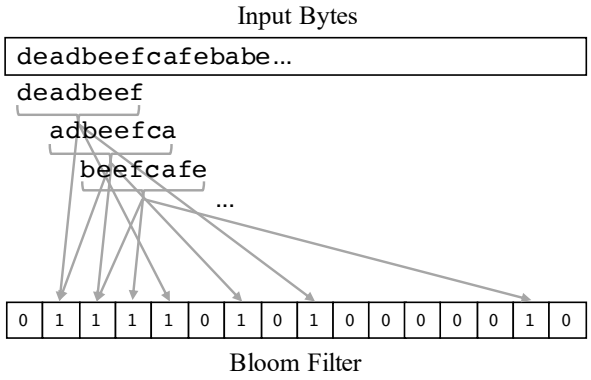



**Figure 7.3:** An example on how transaction input bytes are mapped into a bloom filter.

the filter if all indices of the 6 hash functions are set to one. We use interleaved $n$-grams because the input of a copied transaction might be included at any position in the attacker's input. Once our linear search finds two transactions $T_A$ and $T_V$ with matching inputs, we check whether the following three heuristics hold:

**Heuristic 1:** The sender of $T_A$ and $T_V$ as well as the receiver of $T_A$ and $T_V$ must be different. The receiver of $T_A$ and $T_V$ has to be different to make sure that we only detect displacement attacks that are performed by bot contracts.

**Heuristic 2:** The gas price of $T_A$ must be larger than the gas price of $T_V$.

**Heuristic 3:** We split the transaction input of $T_A$ and $T_V$ into sequences of 4 bytes, and the ratio between the number of the sequences must be at least 25%. This heuristic requires that the byte sequences from $T_V$ conform with at least 25% of the byte sequences of $T_A$ to avoid false positives. Without this restriction, it is very common for transactions with very small inputs to match by chance against transactions with very large inputs.

However, the aforementioned heuristics may not filter out all the benign cases and therefore produce false positives. As a result, we filter out the benign cases by applying a runtime validation on the transaction inputs. The heuristics are still useful and necessary since the validation process is computationally very intensive and the heuristics help us reduces the number of cases to validate and thus save time. To validate that $T_A$ is a copy of $T_V$, we run in a simulated environment first $T_A$ before $T_V$ and then $T_V$ before $T_A$. We report a finding if the number of executed EVM instructions is different across both runs for $T_A$ and $T_V$, as this means that $T_A$ and $T_V$ influence each other. During our experiments, we noted that some bot contracts included code that checks if the miner address of the block that is currently being executed is not equal to zero. We think that the goal of this mechanism could be to prevent transactions from being run locally.

**Limitations.** With more than 11 million blocks and over 1 billion transactions, we were compelled to make trade-offs between efficiency and completeness. To be able to scan the entire blockchain for displacement attacks in a reasonable amount of time, we decided to set a window size of 100 blocks, meaning that we could not detect displacement attacks were an attacker's transaction and a victim's transaction are more than 100 blocks apart. Another limitation is that our heuristics only focus on detecting displacement attacks performed by bot contracts. For example, attackers can also send a transaction directly to the contract that is susceptible to displacement, without going through a bot contract. However, it is difficult for us to distinguish between benign users and attackers in such a case. Therefore, we decided to focus only on detecting bot contracts since a benign user would not use such a contract to perform a transaction to the susceptible contract. Thus, our heuristics might produce false negatives and our results should be considered as a lower bound only.

### 7.3.3 Detecting Insertion

We limit our detection to insertion attacks on decentralized exchanges (DEXes). At the time of writing, we are not aware of any other use case where insertion attacks are applied in the wild. DEXes are decentralized platforms where users can trade their ERC-20 tokens for ether or other ERC-20 tokens via a smart contract. Uniswap is currently the most popular DEX in terms of locked value with 3.15B USD locked[1]. There exist two genres of DEXes, order book-based DEXes and Automated Market Maker-based (AMM) DEXes. While order book-based DEXes match prices based on so-called 'bid' and 'ask' orders, AMM-based DEXes match and settle trades automatically on-chain via a smart contract, without the need of third party service. AMMs are algorithmic agents that follow a deterministic approach to calculate the price of a token. Uniswap, for example, is an AMM-based DEX, which computes for every trade the price of a token using the equation of a Constant Product Market Maker (CPMM):

$$[x] \times [y] = k \tag{7.3}$$

where $[x]$ is the current reserve of token $x$ and $[y]$ is the current reserve of token $y$. Trades must not change the product $k$ of a pair's reserve. Thus, if the underlying token reserves decrease as a trader is buying, the token price increases. The same holds in the opposite direction: if the underlying token's reserve increases while a trader is selling, the token price decreases. Despite being simple, CPMMs are incredibly susceptible to price slippage. Price slippage refers to the difference between a trade's expected price and the price at which the trade is executed. Given the public nature of blockchains, attackers can observe large buy orders before miners pick them up by monitoring the transaction pool. These large buy orders will have a significant impact on the price of a token. Leveraging this knowledge and the fact that miners order transactions based on transaction fees, attackers

[1] https://defipulse.com/

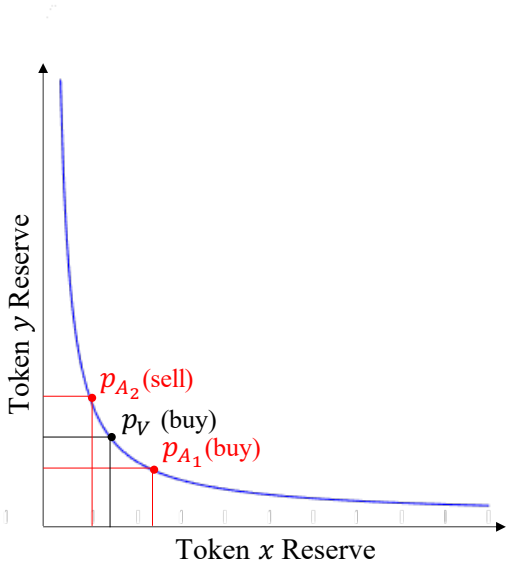



**Figure 7.4:** An illustrative example of an insertion attack on an AMM-based DEX that uses CPMM.

can insert their buy order in front of an observed large buy order and insert a sell order after the observed large buy order to profit from the deterministic price calculation. Figure 7.4 depicts an example of an insertion attack on an AMM-based DEX that uses CPMM. Let us assume that a victim $V$ wants to purchase some tokens at a price $p$. Let us also assume that an attacker $A$ observes $V$'s transaction and sends in two transactions: 1) a buy transaction which also tries to purchase some tokens at a price $p$, but with a gas price higher than $V$, and 2) a sell transaction that tries to sell the purchased tokens, but with a gas price lower than $V$. Since $A$ pays a higher gas price than $V$, $A$'s purchase transaction will be mined first and $A$ will be able to purchase the tokens at price $p$, where $p = p_{A_1}$ (cf. Figure 7.4). Afterwards, $V$'s transaction will be mined. $V$ will purchase tokens at a higher price $p_V$, where $p_V > p_{A_1}$ due to the imbalance in the token reserves (see Equation 3). Finally, $A$'s sell transaction will be mined, for which $A$ will sell its tokens at price $p_{A_2}$, where $p_{A_2} > p_{A_1}$ and therefore $A$ making profit. Our detection algorithm exploits the fact that DEXes depend on the ERC-20 token standard. The ERC-20 token standard defines many functions and events that enable users to trade their tokens between each other and across exchanges. In particular, whenever a token is traded, a so-called `Transfer` event is triggered, and information about the sender, receiver, and the amount is logged on the blockchain. We combine this information with transactional information (e.g., transaction index, gas price, etc.) to detect insertion attacks. We define a transfer event as $E = (s, r, a, c, h, i, g)$, where $s$ is the sender of the tokens, $r$ is the receiver of the tokens, $a$ is the number of transferred tokens, $c$ is the token's contract address, $h$ is the transaction hash, $i$ is the transaction index, and $g$ is the gas price of the transaction. We detect insertion attacks by iterating block by block through all the transfer

events and checking if there are three events $E_{A_1}$, $E_V$, and $E_{A_2}$ for which the following six heuristics hold:

**Heuristic 1:** The exchange transfers tokens to $A$ in $E_{A_1}$ and to $V$ in $E_V$, and the exchange receives tokens from $A$ in $E_{A_2}$. Moreover, $A$ transfers tokens in $E_{A_2}$ that it received previously in $E_{A_1}$. g Thus, the sender of $E_{A_1}$ must be identical to the sender of $E_V$ as well as the receiver of $E_{A_2}$, and the receiver of $E_{A_1}$ must be identical to the sender of $E_{A_2}$ (i.e., $s_{A_1} = s_V = r_{A_2} \wedge r_{A_1} = s_{A_2}$).

**Heuristic 2:** The number of tokens bought by $E_{A_1}$ must be similar to the number of tokens sold by $E_{A_2}$. To avoid false positives, we set a conservative threshold of 1%. Hence, the difference between token amount $a_{A_1}$ of $E_{A_1}$ and token amount $a_{A_2}$ of $E_{A_2}$ cannot be more than 1% (i.e., $\frac{|a_{A_1} - a_{A_2}|}{max(a_{A_1}, a_{A_2})} \leq 0.01$).

**Heuristic 3:** The token contract addresses of $E_{A_1}$, $E_V$, and $E_{A_2}$ must be identical (i.e., $c_{A_1} = c_V = c_{A_2}$).

**Heuristic 4:** The transaction hashes of $E_{A_1}$, $E_V$, and $E_{A_2}$ must be dissimilar (i.e., $h_{A_1} \neq h_V \neq h_{A_2}$).

**Heuristic 5:** The transaction index of $E_{A_1}$ must be smaller than the transaction index of $E_V$, and the transaction index of $E_V$ must be smaller than the transaction index of $E_{A_2}$ (i.e., $i_{A_1} < i_V < i_{A_2}$).

**Heuristic 6:** The gas price of $E_{A_1}$ must be larger than the gas price of $E_V$, and the gas price of $E_{A_2}$ must be less of equal to the gas price of $E_V$ (i.e., $g_{A_1} > g_V \geq g_{A_2}$).

**Limitations.** Our heuristics assume that insertion attacks always occur within the same block. This assumption enables us to check blocks in parallel since we only need to compare transactions within a block. However, this assumption does not always hold in reality, as transactions might be scattered across different blocks during the mining process. Thus, there might exist insertion attacks that were performed across multiple blocks, which our heuristics do not detect and therefore might result in false negatives.

### 7.3.4 Detecting Suppression

In suppression, an attacker's goal is to withhold a victim's transaction by submitting transactions to the network that consume large amounts of gas and fill up the block gas limit such that the victim's transaction cannot be included anymore. There are several ways to achieve this. The naive approach uses a smart contract that repeatedly executes a sequence of instructions in a loop to consume gas. This strategy can either be controlled or uncontrolled. In a controlled setting, the attacker repeatedly checks how much gas is still left and exits the loop right before all gas is consumed such that no out-of-gas exception is raised. In

an uncontrolled setting, the attacker does not repeatedly check how much gas is left and lets the loop run until no more gas is left and an out-of-gas exception is raised. The former strategy does not consume all the gas and does not raise an exception which makes it less obtrusive, while the latter strategy does consume all the gas but raises an exception which makes it more obtrusive. However, a third strategy achieves precisely the same result without running code in an infinite loop. If we think about it, the attacker's goal is not to execute useless instructions but rather to force miners to consume the attacker's gas units to fill up the block. The EVM proposes two ways to raise an error during execution, either through a revert or an assert. The difference between revert and assert is that the former returns the unused gas to the transaction sender, while the latter consumes the entire gas limit initially specified by the transaction sender. Hence, an attacker can exploit this and call an assert to consume all the provided gas with just one instruction. Our goal is to detect transactions that employ one of the three aforementioned suppression strategies: *controlled gas loop*, *uncontrolled gas loop*, and *assert*. We start by clustering for each block all transactions with the same receiver, as we assume that attackers send multiple suppression transactions to the same bot contract. Afterwards, we check the following heuristics for each cluster:

**Heuristic 1:** The number of transactions within a cluster must be larger than one.

**Heuristic 2:** All transactions within a cluster must have consumed more than 21,000 gas units. The goal of this heuristic is to filter out transactions that only transfer value (i.e., ether), but do not execute code.

**Heuristic 3:** The ratio between gas used and gas limit must be larger than 99% for all transactions within the cluster.

If we happen to find a cluster that fulfills the heuristics mentioned above, we check whether at least one of the neighboring blocks (i.e., the previous block and the subsequent block) also contains a cluster that satisfies the same heuristics. We assume that an attacker tries to suppress transactions for a sequence of blocks. Finally, we try to detect if an attacker employs one of three suppression strategies by retrieving and analyzing the execution trace of the first transaction in the cluster. An execution trace consists of a sequence of executed instructions. We detect the first strategy by checking if the transaction did not raise an exception and if the instruction sequence `[GAS, GT, ISZERO, JUMPI]` is executed more than ten times in a loop. This particular instruction sequence checks how much gas is left and jumps towards a different code location, if the amount of gas is lower than a given value. We detect the second strategy by checking if the transaction raised an exception via a revert and if the instruction sequence `[SLOAD, TIMESTAMP, ADD, SSTORE]` is executed more than ten times in a loop. This particular instruction sequence increments a persistent counter residing in storage with the current timestamp in order to consume a large amount of gas.

Finally, we detect the third strategy by checking if the transaction raised an exception via an assert.

**Limitations.** Our heuristics follow two major assumptions. First, we assume that an attacker always sends multiple transactions to the same bot contract. However, an attacker could also just send one transaction and deploy multiple bot contracts for single use. Second, we assume that an attacker always tries to suppress more than just one block. However, an attacker could also just try to suppress one block. While in practice we always observed that attackers tried to suppress multiple blocks and sent multiple transactions as well as reused the same bot contract, it is still possible that some attackers do not follow this pattern and therefore our heuristics might produce false negatives.

## 7.4 Analyzing Frontrunning Attacks

In this section, we analyze the results of our large scale measurement study on detecting frontrunning in Ethereum.

### 7.4.1 Experimental Setup

We implemented our detection modules using Python with roughly 1,700 lines of code[2] We run our modules on the first 11,300,000 blocks of the Ethereum blockchain, ranging from July 30, 2015 to November 21, 2020. All our experiments were conducted using a machine with 128 GB of memory and 10 Intel(R) Xeon(TM) L5640 CPUs with 12 cores each and clocked at 2.26 GHz, running 64 bit Ubuntu 16.04.6 LTS.

### 7.4.2 Validation

Since our work is the first to systematically study the three different types of frontrunning by leveraging historical blockchain data on such a large scale, we are missing a ground truth against which we can compare our results. Our goal was therefore to design very precise and rather conservative heuristics that might yield false negatives, but no false positives. We started with a rather liberal definition of our heuristics and did several iterations, where we regularly checked for outliers and tried to tighten the heuristics after each iteration whenever we discovered false positives in our preliminary results. After finding no more false positives we ran our final experiments, which resulted in over 200K transactions being labeled as either displacement, insertion, or suppression frontrunning attacks. Since checking all of these 200K transactions manually is extremely cumbersome, we decided to select a random sample of 100 findings for each type of frontrunning attack and manually check them for false

[2]Code and data are publicly available at: https://github.com/christoftorres/Frontrunner-Jones.

**Table 7.1:** Distributions for displacement attacks.

| | Cost (USD) | Profit (USD) | Gas Price $\Delta$ (Gwei) | Block $\Delta$ |
|---|---|---|---|---|
| mean | 14.28 | 1,537.99 | 0.43 | 0.78 |
| std | 18.25 | 7,162.80 | 2.65 | 2.37 |
| min | 0.01 | 0.00 | 0.00 | 0.00 |
| 25% | 4.36 | 1.14 | 0.00 | 0.00 |
| 50% | 9.48 | 158.53 | 0.00 | 0.00 |
| 75% | 16.64 | 851.04 | 0.00 | 0.00 |
| max | 311.69 | 223,150.01 | 52.90 | 19.00 |

positives. For displacement, we tried to reverse engineer the code of the identified bot contract to see if the code was proxying the transaction input to a specified contract destination. For insertion, we checked if the two reported attacker transactions and the whale transaction were indeed buying or selling the exact same token via the same exchange. Finally, for suppression, we tried to reverse engineer the reported bot contract and to check if the contract would probe who is the last purchaser of a ticket of a specific lottery or gambling contract and try to consume the entire gas in case the last purchaser was a specific address. Following these steps, our manual validation did not reveal any false positives. However, as already mentioned previously, our heuristics have some limitations which might result in false negatives. Hence, all the results presented in this chapter should be interpreted only as lower bounds, and they might only show the tip of the iceberg.

### 7.4.3 Analyzing Displacement

**Overall Results.** We identified a total of 2,983 displacement attacks from 49 unique attacker accounts and 25 unique bot contracts. Using the graph analysis defined in Section 7.3.1 we identified 17 unique attacker clusters.

**Profitability.** We compute the gain of an attacker $A$ on each displacement attack by searching how much ether $EOA_A$ receives among the internal transactions triggered by $T_A$. Additionally, we obtain the profit by subtracting the attack cost from the gain, where cost is defined solely by the fees of $T_A$. Finally, for each attack we convert the ether cost and profit into USD by taking the conversion rate valid at the time of the attack.

**Attacks.** We can see in Table 7.1 the distribution of each variable we collected per displacement attack. The cost and the profit do not appear to be very high for most of the attacks, but the distributions of both variables present very long tails to the right. Additionally, we compute the Gas Price $\Delta$ as the gas price of $T_A$ minus the gas price of $T_V$. This value indicates how much the attacker $A$ is willing to pay to the miners so they execute $T_A$ before $T_V$.

**Table 7.2:** Distributions for displacement attacker clusters.

| | Cost (USD) | Profit (USD) | Attacks | Attacker Accounts | Bot Contracts |
|---|---|---|---|---|---|
| mean | 2,505.09 | 269,872.45 | 175.47 | 2.88 | 1.47 |
| std | 9,776.51 | 1,005,283.40 | 555.03 | 3.89 | 0.80 |
| min | 0.05 | 0.01 | 1.00 | 1.00 | 1.00 |
| 25% | 0.14 | 3.53 | 1.00 | 1.00 | 1.00 |
| 50% | 3.98 | 726.70 | 5.00 | 1.00 | 1.00 |
| 75% | 65.78 | 4,670.94 | 8.00 | 3.00 | 2.00 |
| max | 40,420.63 | 4,152,270.01 | 2249.00 | 16.00 | 3.00 |

Table 7.1 shows that most of the attacks contain a very small gas price difference in Gwei (and cannot be represented with only two digits of precision), but there are very extreme cases with a difference close to 50 Gwei. Furthermore, we compute the Block $\Delta$ to indicate how many blocks are between the execution of $T_A$ and $T_V$. Again we can see in Table 7.1 that for most of the attacks, both transactions were executed in the same block, but there are some extreme cases with a long block distance of 19 blocks.

**Attacker Clusters.** Each of the 17 identified clusters contains bot accounts with different bytecode, with the exception of one cluster that contains three bot accounts with the exact same bytecode. Table 7.2 presents the distribution of each attacker cluster variable. The first variable describes profit, where we can see that a single attacker mounted 2,249 attacks making an accumulated profit of more than 4.1M USD while spending over 40K USD in transaction fees. We can also see that the attacker used 16 different accounts and 3 different bots to mount its attacks. The minimum amount of profit that an attacker made with displacement is 0.01 USD. Overall, the average number of attacks per attacker cluster is 175.47 attacks, using 2.88 accounts and 1.47 bots. However, we also observe from the distribution that at least half of the attackers only use one account and one bot contract.

### 7.4.4 Analyzing Insertion

**Overall Results.** We identified a total of 196,691 insertion attacks from 1,504 unique attacker accounts and 471 unique bot contracts. Using the graph analysis defined in Section 7.3.1 we identified 98 unique attacker clusters.

**Profitability.** We compute the cost for each attack as the sum of the amount of ether an attacker spent in $T_{A_1}$ and the fees imposed by transactions $T_{A_1}$ and $T_{A_2}$. Additionally, we compute the profitability of an attack as the amount of ether an attacker gained in $T_{A_2}$ minus the cost. Finally, for each attack we convert the ether cost and profit into USD by taking the conversion rate valid at the time of the attack.

**Table 7.3:** Distributions for insertion attacks.

| | Cost (USD) | Profit (USD) | Gas Price $\Delta_1$ (Gwei) | Gas Price $\Delta_2$ (Gwei) |
|---|---|---|---|---|
| mean | 19.41 | 65.05 | 407.63 | 3.88 |
| std | 51.15 | 233.44 | 1,897.47 | 137.12 |
| min | 0.01 | -10,620.61 | 0.00 | 0.00 |
| 25% | 4.09 | 7.86 | 0.00 | 0.00 |
| 50% | 7.74 | 24.07 | 5.25 | 0.00 |
| 75% | 15.23 | 62.92 | 74.10 | 0.00 |
| max | 1,822.22 | 20,084.01 | 76,236.09 | 27,396.63 |

**Attacks.** We can see in Table 7.3 the distribution of each variable we collected per insertion attack. The cost and the profit do not appear to be very high for most of the attacks, but the distributions of both variables present very long tails to the right. Note that the profit also present very large negative values to the left, meaning that there are extreme cases of attackers losing money. Additionally, we compute the Gas Price $\Delta_1$ and Gas Price $\Delta_2$ as the gas price of $T_{A_1}$ minus the gas price of $T_V$, and the gas price of $T_V$ minus the gas price of $T_{A_2}$ respectively. This value indicates how much the attacker $A$ is willing to pay to the miners so they execute $T_{A_1}$ before $T_V$ and also if $T_{A_2}$ can be executed after $T_V$. Table 7.3 shows that 25% of the attacks contain a very small Gas Price $\Delta_1$ in Gwei (and cannot be represented with only two digits of precision), but that half or more paid a significant difference, reaching some extreme cases of more than 76K Gwei. For Gas Price $\Delta_2$ most of the attacks have a very small value, but there are extreme cases, which mean that some attacks are targeting transactions with very high gas prices.

**Gas Tokens.** We analyzed how many attacks were mounted using gas tokens. Gas tokens allow attackers to reduce their gas costs. We found that 63,274 (32.17%) of the insertion attacks we measured were performed using gas tokens. Of these, 48,281 (76.3%) attacks were mounted using gas tokens only for the first transaction $T_{A_1}$, 1,404 (2.22%) attacks were mounted by employing gas tokens only for the second transaction $T_{A_2}$, and 13,589 (21.48%) attacks were mounted by employing gas tokens for both transactions $T_{A_1}$ and $T_{A_2}$. We also found that 24,042 (38%) of the attacks used GST2, 14,932 (23.6%) used ChiToken, and 24,300 (38.4%) used their own implementation or copy of GST2 and ChiToken.

**Exchanges and Tokens.** We identified insertion attacks across 3,200 different tokens on four exchanges: Bancor, Uniswap V1, Uniswap V2, and SushiSwap. Figure 7.5 depicts the weekly average of daily insertion attacks per exchange. The first AMM-based DEX to be released on Ethereum was Bancor in November 2017. We observe from Figure 7.5 that the first insertion attacks started in February 2018, targeting the Bancor exchange. We also see that the number of insertion attacks increased tremendously with the rise of other DEXes,

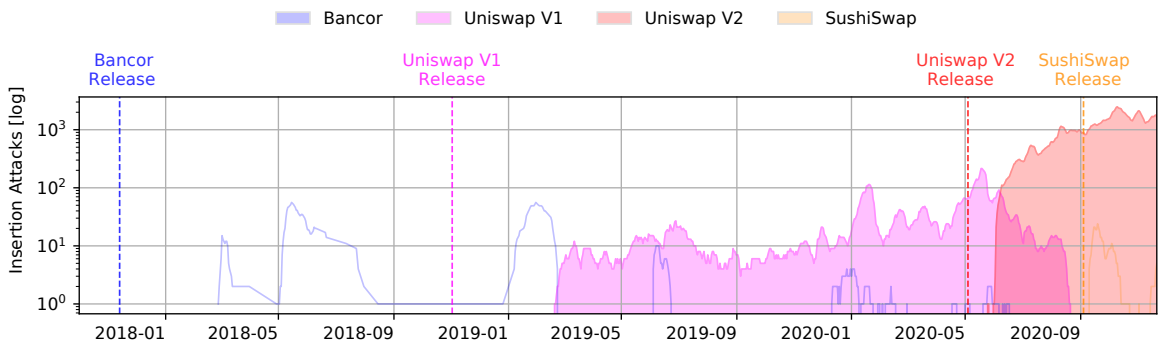



**Figure 7.5:** Weekly average of daily insertion attacks per decentralized exchange.

such as Uniswap V1 and Uniswap V2. While it took 3 months for attackers to launch their first insertion attacks on Uniswap V1, it only took 2 weeks to launch attacks on Uniswap V2 and 5 days to launch attacks on SushiSwap. This is probably due to the core functionality of Uniswap V1 and Uniswap V2 being the same and that SushiSwap is a direct fork of Uniswap V2. Thus, for attackers it was probably straightforward to take their existing code for Uniswap V1 and adapt it to attack Uniswap V2 as well as SushiSwap. The peak of insertion attacks was on October 5, 2020, with 2,749 daily attacks. We measured in total 3,004 attacks on Bancor, 13,051 attacks on Uniswap V1, 180,185 attacks on Uniswap V2, and 451 attacks on SushiSwap. Table 7.4 shows the different combinations of exchanges that attackers try to frontrun. We see that most of the attackers focus on attacking Uniswap V2, with 72 attacker clusters (73.47%). We also see that 92.86% of the attackers only focus on attacking one exchange. Moreover, we observed one attacker that attacked all the 4 exchanges, 2 attackers that attacked Uniswap V1 and Uniswap V2, and 4 attackers that attacked Uniswap V2 and SushiSwap. The latter is expected since SushiSwap is a direct fork of Uniswap V2. Hence, the attackers can reuse their code from Uniswap V2 to attack SushiSwap. What is interesting though, is the fact that no attacker is attacking only SushiSwap, we see that attacker always attack SushiSwap in conjunction to another exchange.

**Table 7.4:** Exchange combination count by attacker cluster.

| Exchange Combination | Attacker Clusters |
|---|---|
| Uniswap V2 | 72 |
| Uniswap V1 | 16 |
| SushiSwap, Uniswap V2 | 4 |
| Bancor | 3 |
| Uniswap V1, Uniswap V2 | 2 |
| Bancor, SushiSwap, Uniswap V1, Uniswap V2 | 1 |

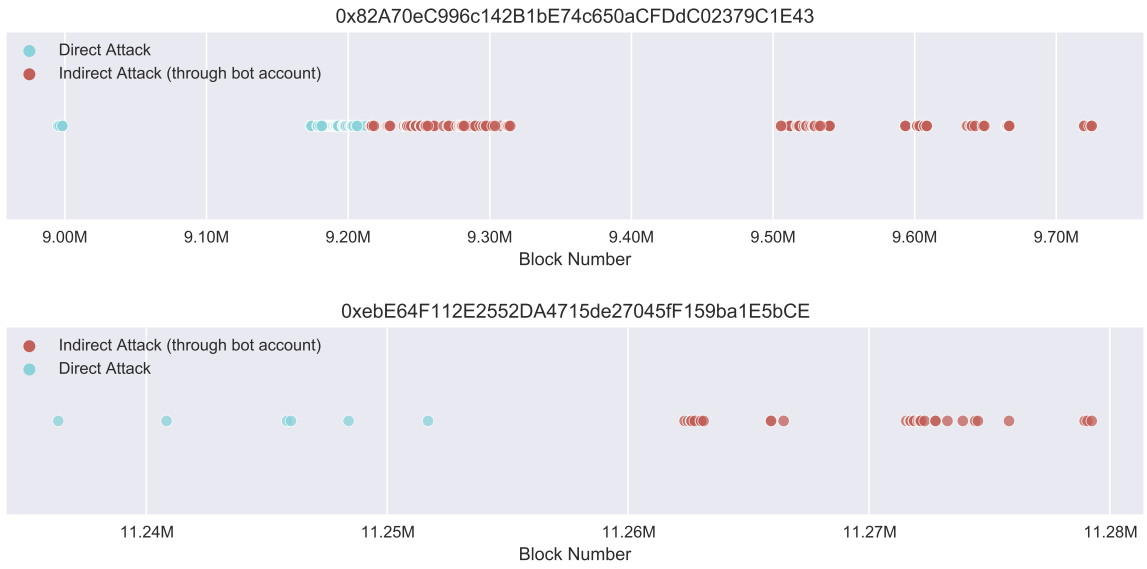



**Figure 7.6:** Two examples of attackers changing their strategies over time from direct attacks (i.e., using directly an exchange) to indirect attacks (i.e., using a bot contract).

**Attack Strategies.** In 186,960 cases (95.05%) the attackers sold the exact same amount of tokens that they purchased. Thus, an easy way to spot insertion attacks on decentralized exchanges is to check for two transactions that have the same sender and receiver, and where the first transaction buys the same amount of tokens that the second transaction sells. However, some attackers try to obscure their buy and sell transactions by using different sender accounts. We found 86,038 cases of attacks (43.74%) where attackers used a different sender address to buy tokens than to sell tokens. Moreover, besides trying to hide their sender accounts, attackers also try to hide in some cases the addresses of their bot contracts by using proxy contracts to forward for instance the call to buy tokens to the bot contracts. To the outsider it will look like two transactions with different receivers. We found only 5,467 cases (2.78%) where the attackers are using proxy contracts to disguise calls to their bot contracts. Insertion is the only attack type for which our heuristics can detect attacks that do not employ bot contracts. For these cases, the attacker accounts call the DEXes directly. From all the insertion attacks we detected, only 2,673 cases (0.01%) fall in this category of direct attacks. We included these attacks in most of the results, but we do not count them for the cluster computation since we cannot link the corresponding attacker accounts to any bot contract. Figure 7.6 highlights examples of two accounts that changed their attack strategy over time. The attackers initially performed their attacks by calling directly the smart contract of exchanges, but then switched to bot contracts over time.

**Attacker Clusters.** Among the 98 attacker clusters that we identified, many of the bot contracts share the same bytecode. The most extreme case is an attacker cluster that

**Table 7.5:** Distributions for insertion attacker clusters.

| | Cost (USD) | Profit (USD) | Attacks | Attacker Accounts | Bot Contracts |
|---|---|---|---|---|---|
| mean | 38,807.63 | 130,246.93 | 1979.78 | 14.87 | 4.81 |
| std | 135,352.00 | 462,464.36 | 6053.68 | 90.59 | 10.09 |
| min | 0.98 | -2,319.42 | 1.00 | 1.00 | 1.00 |
| 25% | 43.84 | -9.78 | 4.25 | 1.00 | 1.00 |
| 50% | 419.74 | 691.48 | 68.50 | 2.00 | 2.00 |
| 75% | 3,510.94 | 8,350.46 | 529.25 | 3.00 | 4.00 |
| max | 686,850.37 | 2,262,411.95 | 39162.00 | 891.00 | 80.00 |

contains 80 bot contracts and all of them have the same bytecode. We find that attackers were already able to make an accumulated profit of over 13.9M USD. From Table 7.5, we see that an attacker makes on average a profit of over 130K USD per attacker cluster. Moreover, the average profit per attack is 78.72 USD, whereas the median profit is 28.80 USD. The largest profit that has been made with a single attack was 20,084.01 USD. However, not all the attacks were successful in terms of profit. We count 19,828 (10.08%) attacks that resulted in an accumulated loss of roughly 1.1M USD. The largest loss that we measured was 10,620.61 USD. The average loss is 56.93 USD per attack and the median loss is 14.26 USD per attack. Thus, the average loss is still lower than the average profit, meaning that insertion attacks are profitable despite bearing some risks.

**Competition.** We found among our detected results 5,715 groups of at least two insertion attacks that share the same block number, victim transaction and exchanged token but with different attackers. Included in those groups, we found 270 cases where at least two of the attackers targeting the same victim belong to the same attacker cluster. To explain this phenomenon, we have three hypothesis. The first one is that an attacker would not interfere with its own attacks, hence, our attacker clustering mechanism is incorrect. Since our methodology is based on heuristics and we have no ground truth to validate them, we could expect to find occasional errors. However, since the heuristics are simple and reasonable enough, we also consider the next two hypothesis. The second one is that some attackers might not be clever enough to coordinate multiple agents working in parallel, and the self-interference could be an accident. And third, the parallel attacks could be a tactic to split the movements of funds into smaller amounts to avoid becoming the target of other attackers. For example, we found two instances where attackers became victims at the same time, namely accounts `0x5e334032Fca55814dDb77379D8f99c6eb30dEa6a` and `0xB5AD1C4305828636F32B04E5B5Db558de447eAff` in blocks 11,190,219 and 11,269,029, respectively.

### 7.4.5 Analyzing Suppression

**Overall Results.** We identified a total of 50 suppression attacks originated from 98 attacker accounts and 30 bot contracts. From these entities, we identified 5 unique attacker clusters using the graph analysis defined in Section 7.3.1.

**Rounds, Success, and Failure.** In this section we define a suppression attack as a sequence of rounds. Each round starts with an investment transaction that sends ether to the victim's contract, which is added to a prize pool. The round then continues with a sequence of one or more stuffing transactions. When another participant interrupts the stuffing sequence by sending a new investment transaction, the participant becomes the new potential winner of the prize pool. This event terminates the round in a failure state, because the attacker cannot claim the prize anymore. Otherwise, if an interruption never occurs and the attacker can eventually claim the competition prize, the round is terminated with a success status. Thus, we define the status of an entire suppression attack as the status of the last round in the corresponding sequence of rounds. From the 50 suppression attacks we identified, 13 were successful and 37 failed.

**Suppression Strategies.** In Table 7.6 we show the distribution of suppression strategies split by successful and failed attacks. We see that although the assert strategy is the most popular one, it is not the most successful one. The controlled gas loop strategy seems to be the most successful in terms of attacks.

**Table 7.6:** Suppression strategies.

| Suppression Strategy | Attacks | Successful | Failed |
|---|---|---|---|
| Assert | 20 | 2 | 18 |
| Controlled Gas Loop | 18 | 8 | 10 |
| Uncontrolled Gas Loop | 12 | 3 | 9 |

**Profitability.** In a suppression attack, the profit of the attacker $A$ is defined by the accumulated ether in the prize pool of the suppressed contract. Note that the attack only obtains the prize if it succeeds. Additionally, we subtract from the profit the attack cost which is defined by the sum of the initial investment on each round, and the accumulated fees of all the related transactions $T_{A_i}$. Finally, for each attack we convert the ether cost and profit into USD by taking the conversion rate valid at the time of the attack.

**Attacks.** We can see in Table 7.7 the distribution of each variable we collected per suppression attack. An interesting result is that at least 75% of the attacks generate big losses. However, there are also extreme cases with huge profits. Hence, we could say that the

suppression attacks are very risky but that they can also yield high rewards. Along with the price and cost, we also count the number of rounds, blocks and transactions every attack contains. We can observe, as expected in Table 7.7, how all these metrics grow together with the cost. A suppression attack lasts on average 6.62 rounds and an attacker stuffs on average 29.70 blocks with an average of 182.70 transactions.

**Table 7.7:** Distributions for suppression attacks.

| | Cost (USD) | Profit (USD) | Rounds | Blocks | Transactions |
|---|---|---|---|---|---|
| mean | 2,349.65 | 20,725.24 | 6.62 | 29.70 | 182.70 |
| std | 3,331.21 | 113,598.58 | 12.86 | 50.77 | 456.91 |
| min | 4.67 | -10,741.12 | 1.00 | 2.00 | 6.00 |
| 25% | 221.87 | -1,893.26 | 1.00 | 4.00 | 12.50 |
| 50% | 896.68 | -284.81 | 2.00 | 10.00 | 33.50 |
| 75% | 2,719.69 | -14.93 | 4.75 | 21.50 | 88.75 |
| max | 10,741.12 | 791,211.86 | 66.00 | 233.00 | 2,664.00 |

**Attacker Clusters.** We identified 5 attacker clusters. Among the attacker clusters, we found only two pairs of bot contracts sharing the same bytecode. From Table 7.8, we can see that the average profit per attacker cluster is 207,252.36 USD and that the largest profit made by an attacker cluster is over 777K USD. However, we also see that mounting suppression attacks is expensive with an average of 23,496.52 USD, but still profitable with an average profit of 207,252.36 USD. Also, we find that attackers mount on average 10 attacks and use on average around 19 attacker accounts and 6 bot contracts. There is one case where an attacker was responsible for mounting 18 different attacks using 42 different accounts and 14 different bots.

**Table 7.8:** Distributions for suppression attacker clusters.

| | Cost (USD) | Profit (USD) | Attacks | Attacker Accounts | Bot Contracts |
|---|---|---|---|---|---|
| mean | 23,496.52 | 207,252.36 | 10.00 | 19.60 | 6.00 |
| std | 20,520.87 | 323,613.48 | 7.65 | 13.67 | 5.24 |
| min | 46.00 | -46.00 | 1.00 | 6.00 | 1.00 |
| 25% | 14,836.39 | 19,274.31 | 3.00 | 12.00 | 2.00 |
| 50% | 21,592.43 | 115,241.45 | 12.00 | 18.00 | 5.00 |
| 75% | 25,054.40 | 124,243.35 | 16.00 | 20.00 | 8.00 |
| max | 55,953.37 | 777,548.67 | 18.00 | 42.00 | 14.00 |

**Table 7.9:** List of contracts that were victims of suppression attacks.

| Contract Name | Attacks | Rounds | Transactions | Attackers | Bot Contracts | Attacker Clusters |
|---|---|---|---|---|---|---|
| Last Winner | 16 | 20 | 304 | 27 | 5 | 2 |
| FoMo3Dlong | 12 | 188 | 5875 | 81 | 8 | 4 |
| Peach Will | 6 | 52 | 1105 | 26 | 5 | 2 |
| ERD | 3 | 3 | 207 | 20 | 2 | 1 |
| ETH CAT | 2 | 23 | 929 | 20 | 2 | 1 |
| Escape plan | 2 | 3 | 67 | 20 | 2 | 1 |
| SuperCard | 1 | 25 | 319 | 17 | 1 | 1 |
| Mobius2Dv2 | 1 | 4 | 82 | 19 | 1 | 1 |
| Star3Dlong | 1 | 3 | 66 | 6 | 1 | 1 |
| FDC | 1 | 3 | 44 | 18 | 1 | 1 |
| F3DPRO | 1 | 1 | 41 | 18 | 1 | 1 |
| FomoXP | 1 | 3 | 39 | 19 | 1 | 1 |
| EFS | 1 | 1 | 33 | 16 | 1 | 1 |
| The rabbit | 1 | 1 | 15 | 13 | 1 | 1 |
| RichKey | 1 | 1 | 9 | 8 | 1 | 1 |

**Competition.** We found that suppression attacks only targeted 15 unique contracts, which are listed in Table 7.9. We can see that only the contracts Last Winner, FoMo3Dlong, and Peach Will were targeted by different attacker clusters. We searched through all the attacks for blocks where any of these three contracts were the victims and more than one attacker cluster was targeting the same victim. We found only one case where bot contract `0xDd9fd6b6F8f7ea932997992bbE67EabB3e316f3C` started an attack interrupting another attack from bot contract `0xd037763925C23f5Ba592A8b2F4910D051a57A9e3` targeting Last Winner on block 6,232,122.

### 7.4.6 Summary

In the following, we summarize our previous findings and compare the different types of frontrunning attacks in terms of structure, competition, cost, profit, bot triggers, bot activity, and trends.

**Structure.** As shown in Figure 7.2, the difference between each attack type is the number of transactions the attacker employs and where the attacker places them in a block relative to the victim. For displacement, the attacker needs to place only one transaction before the victim. For insertion, the attacker creates a sandwich of two transactions around the victim's transaction. Finally, for suppression, the attacker must delay the victim's transaction with one or more transactions.

**Competition.** Attackers can interrupt each other depending on the structure of the attack. For displacement, the attacker only sends one transaction before the victim, so the only way

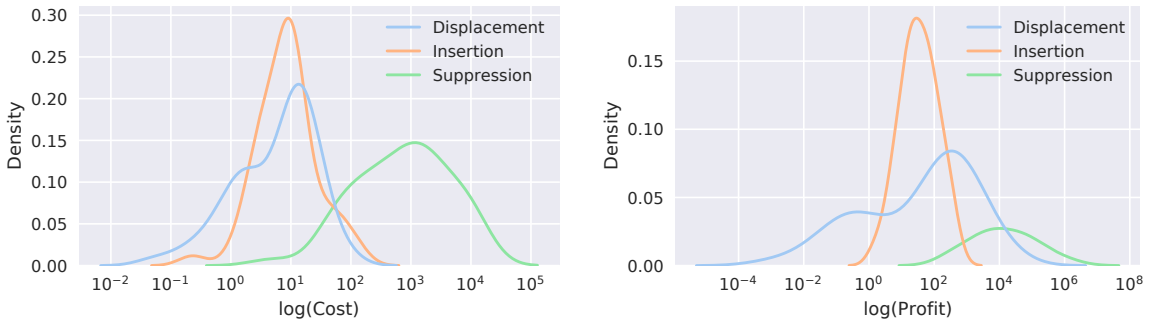



**Figure 7.7:** Cost (left) and profit (right) distributions in logarithmic scale.

for another attacker to interrupt the attack is to place another transaction before the attacker (i.e., with a higher gas price than the victim and the attacker). Moreover, note that the second attacker could be aiming at the same victim or could be considering the first attacker as the victim. In insertion, competition is more complex since one or more transactions can interfere between the three transactions of a sandwich. Additionally, one attacking transaction can take the role of the victim in another sandwich (i.e., when the attacking transaction moves so many funds that it is considered a whale transaction for other attackers). Finally, the suppression case is even easier to interrupt given the number of transactions it involves over an extended range of blocks. Interestingly, the results from our heuristic show that interruptions from regular lottery participants caused most of the failed attacks.

**Cost.** We present the distribution of attack cost for each attack type in Tables 7.1, 7.3 and 7.7. In Figure 7.7 (left), we present the three cost distributions all together. We employed a logarithmic scale on the x-axis because the high density of displacement cost around zero as well as the large dispersion of suppression costs, makes the visualization hard to interpret. However, using this logarithmic scale, we cannot visualize the actual cost, but we can see that suppression attacks tend to be more expensive and have more diverse costs.

**Profit.** Similar to the cost, we present the distribution of attack profit for each attack type in Tables 7.1, 7.3 and 7.7. In Figure 7.7 (right), we present the three profit distributions all together. Similar to the cost, we employed a logarithmic scale on the x-axis because the high density of insertion profit around zero as well as the large dispersion of suppression profits, makes the visualization hard to interpret. In this scale, we cannot visualize the actual profit, but we can see that displacement attacks tend to be more profitable than insertion attacks and that suppression attacks tend to be more profitable than the other two. Additionally, the displacement profit distribution seems to have two modes.

**Bot Triggers.** Bots are triggered by transactions that appear in the pool of pending transac-

tions, which on the other hand reflects user activity. Thus, bots respond to actions performed by human users. For instance, in the case of displacement these triggers can be users accessing smart contracts that do not have proper access control. For insertion, bots are typically triggered by large trades that users commit on decentralized exchanges. Finally, for suppression, bots are triggered when the prize pool of a lottery or gambling contract has reached a significant amount of value, which makes running a suppression attack lucrative.

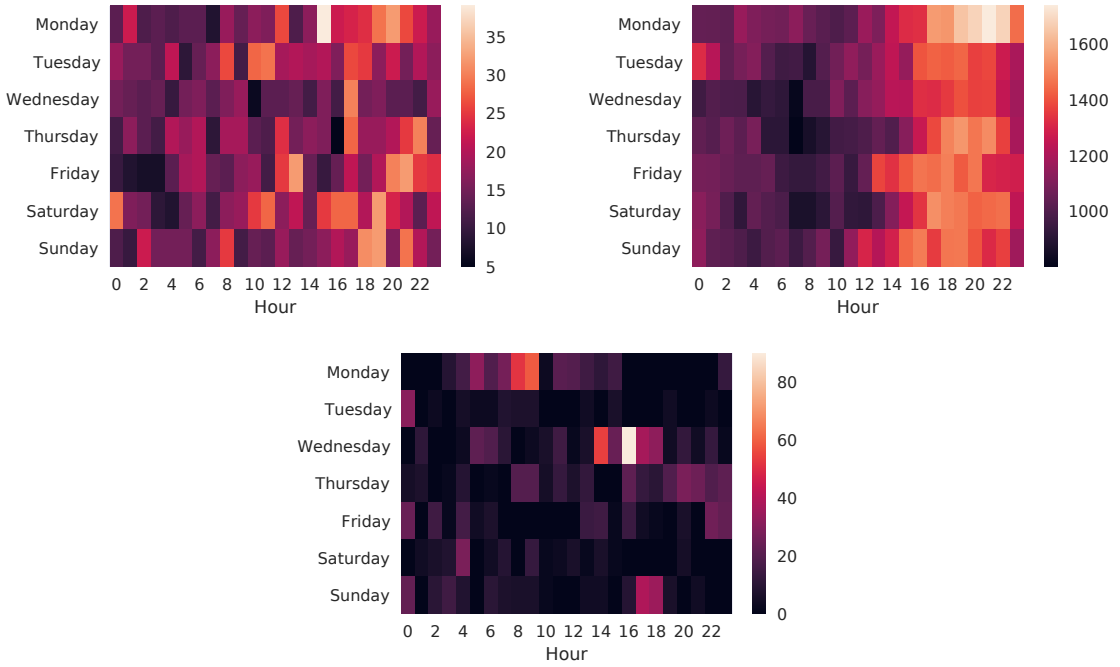



**Figure 7.8:** Number of attacks by weekday and hour for displacement (top left), insertion (top right), and suppression (bottom), following the UTC timezone.

**Bot Activity.** Figure 7.8 describes the number of attacks by weekday and hour for displacement, insertion, and suppression, respectively, using Coordinated Universal Time (UTC) as timezone. We can see that the distribution for displacement appears to be random. For insertion, our results indicate higher bot activity overlapping with evening hours in the northern hemisphere, with highest activity between five and midnight. One plausible explanation is that transactions vulnerable to insertion attacks correspond to human-initiated trading on the blockchain. The evening activity can be explained by the fact that most people have more time to do trading on decentralized exchanges at the end of the day (e.g., after work or after dinner). However, as discussed previously, user activity triggers bots, and users belong to different parts of the world with different timezones, so it is hard to draw any conclusions. We leave it to future work to validate whether the increase of trading activity on decentralized exchanges has led to the increase of insertion frontrunning attacks and whether most users engage in trading activities at the end of the day. Finally, there is a slightly higher activity on Wednesdays for suppression, but we are unsure if the reason depends on a particular

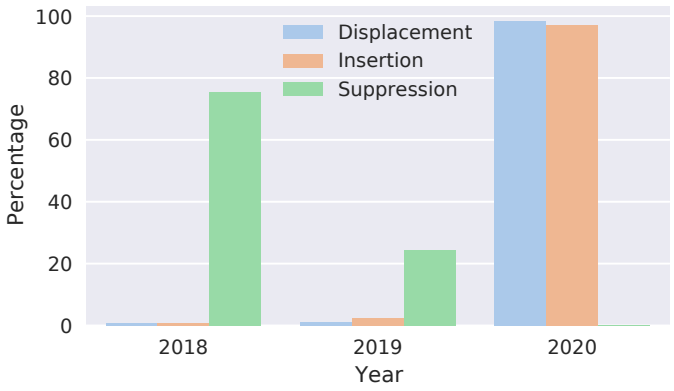



**Figure 7.9:** Percentage of attacks by year.

lottery (e.g., advertisement) or if this is just a coincidence due to our small sample size of detected suppression attacks.

**Trends.** The number of attacks has a very different magnitude for each attack type: 2K for displacement, 197K for insertion and only 50 for suppression. This difference makes it hard to visualize how the amount of attacks changes overtime for all the attacks at the same time. For that reason, in Figure 7.9, we present the percentage of attacks by year for each type of attack. We cannot compare the absolute values in the y-axis, but we can see how suppression attacks decreased over the years, and how both displacement and insertion mostly appear in 2020.

## 7.5 Discussion

In this section, we discuss the implications of frontrunning and why existing mitigation techniques are not effective.

### 7.5.1 Implications of Frontrunning

Daian et al. [150] emphasize that miners could engage in frontrunning activities to maximize or increase their profits. This will most likely be the case when EIP-2878 becomes accepted and the current static block award drops from 2 ETH to 0.5 ETH [185]. However, at the moment miners are already profiting indirectly from frontrunning activities performed by non-miners, since the high gas prices that those non-miners pay end up being for the miners in the form of transaction fees. Thus, miners are incentivized to allow frontrunning. Our results show that miners already earned more than 300K USD from transaction fees payed by the attackers performing insertion frontrunning attacks. While transaction fees in January 2018 only represented 9% of the monthly revenue of a miner, in January 2021 nearly 40% of the monthly revenue came from transaction fees [212]. Thus, besides attackers, we also

concluded that miners profit from frontunning attacks. However, attackers and miners are not the only entities that profit from frontrunning. Take the example of Uniswap. In general, Uniswap takes a 0.3% fee on every transaction. This fee is divided between the liquidity providers, proportionally to their share. For example, if you provide 50% of the liquidity, then you will earn 50% of the collected fee. Thus, liquidity providers profit from every trade performed by frontrunners. However, frontrunning attacks can also have some severe implications for normal users in general. For instance, due to multiple attackers trying to frontrun other attackers via gas price auctions, they temporarily push the average gas prices of the network and force users that do not engage in frontrunning to either pay higher transaction fees or wait longer for their transactions to be mined. This becomes a vicious circle where once again the miners profit from the fact that benign users have to pay higher transaction fees for their transactions to be mined. Thus, the more attackers engage in frontrunning, the more it will have an impact on benign users. Another issue is suppression, which prevents blocks to be used or filled in an optimal way. Ethereum already struggles with a low transaction throughput [161] and suppression attacks only amplify the issue. Suppression attacks can cause the network to congest and decentralized applications to stop working properly.

### 7.5.2 Limitations of Existing Mitigations

There are currently two main reasons why frontrunning is conceivable on public blockchains such as Ethereum. The first reason is the lack of transaction confidentiality. Every node in the network, not just miners, can observe all the transactions in the clear before they are mined. The fact that transactions are transparent to everyone is undoubted one of the major advantages of a public blockchain, however the content and purpose of a transaction should only be visible to everyone once it has been mined. The second reason is the miner's ability to arbitrarily order transactions. This puts a lot of power into the hands of miners. Miners can decide to censor transactions or change the order of transactions such that they make the most profit. The idea to order transactions based on the gas price sounds rational at first, however this also introduces determinism in a way that can be manipulated by outsiders. A suitable mitigation technique must address these two issues, but it must also be efficient in terms of costs for the users, provide fair incentives for miners to continue mining transactions, and be adoptable by everyone and not just by a special group of participants. In our study, we observed that most frontrunning is happening on DEXes, since the risk of failure is low compared to the amount of profit that can be made. Uniswap, the DEX most affected by frontrunning, is aware of the frontrunning issue and proposes a slippage tolerance parameter that defines how distant the price of a trade can be before and after execution. The higher the tolerance, the more likely the transaction will go through, but also the easier it will be for an attacker to frontrun the transaction. The lower the tolerance, the more likely the transaction will not go through, but also the more difficult it will be for an attacker to frontrun the

transaction. As a result, Uniswap's users find themselves in a dilemma. Uniswap suggests by default a slippage tolerance of 0.5% in order to minimize the likelihood that users become victims of frontrunning. However, in this work we prove that the slippage tolerance does not work as we measured over 180K attacks against Uniswap. Hence, other mitigations to counter frontrunning are needed. Bentov et al. [143] present TESSERACT, an exchange that is resistant to frontrunning by leveraging a trusted execution environment. However, their design follows a centralized approach and requires users to have hardware support for trusted execution. Breidenbach et al. [88] proposed LibSubmarine[89], an enhanced commit-and-reveal scheme to fight frontrunning. However, in the case of Uniswap, LibSubmarine would require three transactions to perform a single trade, making it cumbersome and relatively expensive for users to trade.

## 7.6 Related Work

Daian et al. researched frontrunning attacks from an economical point of view by studying gas price auctions [150]. Moreover, by modeling actions of bots using game theory, and framing the problems in terms of a Nash equilibrium for two competing agents, the authors demonstrated that DEXes are severely impacted by two main factors: the high latency required to validate transactions, which opens the door to timing attacks, and secondly the miner driven transaction prioritization based on miner extractable value. The mix of these two factors leads to new security threats to the consensus-layer itself, independent of already existing ones [36, 25]. However, the authors only focused on detecting frontrunning on DEXes and in real time, without scanning the entire blockchain history for evidence of frontrunning. Our work builds on the taxonomy defined by Eskandari et al. [151], which introduces three different types of frontrunning: displacement, insertion, and suppression. Despite illustrating a few concrete examples and discussing several mitigation techniques, the authors did not analyze the prevalence of frontrunning attacks in the wild. Zhou et al. [226] estimated the potential effect of frontrunning on DEXes but limited their analysis only to insertion attacks on a single exchange. Their study estimated the theoretical profit that could have been made if users would have engaged in fruntrunning attacks, but did not back their conclusion with real observed data. Compared to their work, we perform real world measurements not only for insertion attacks, but for the complete spectre of attack types (i.e., displacement, insertion, and suppression). Besides studying frontrunning, a few mitigation techniques have also been proposed to counter frontrunning. For instance, Kelkar et al. proposed a consensus protocol to achieve transaction order-fairness [184]. Breidenbach et al. [88] proposed LibSubmarine[89], an advanced commit-and-reveal scheme to fight frontrunning at the application layer. Bentov et al. [143] present TESSERACT, an exchange that is resistant to frontrunning by leveraging a trusted execution environment. Finally, Kokoris et al. [114] describe CALYPSO, a blockchain that is resistant to frontrunning due to private

transactions. Unfortunately, none of these techniques are broadly adopted as they are either not compatible with the Ethereum blockchain or because they are too costly. Another important side-effect of decentralized finance is the emergence of flash loans [224]. Wang et al. [197] discuss a methodology to detect flash loans using specific patterns and heuristics. We leave it to future work to study the implications of flash loans in the context of frontrunning.

## 7.7 Conclusion

In this chapter, we investigated the prevalence of frontrunning attacks in Ethereum, by presenting a methodology to efficiently measure the three different types of frontrunning attacks: *displacement*, *insertion*, and *suppression*. Our large-scale analysis on the Ethereum blockchain identified 199,725 attacks with an accumulated profit of over 18.41M USD for the attackers. We discussed implications of frontrunning and found that miners profit from frontrunning practices. Miners already made an accumulated passive income of more than 300K USD only from transaction fees payed by frontrunners. Overall, we can conclude that our results shed some light on the predatory actions of the creatures hiding inside Ethereum's dark forest, and we provide evidence that frontrunning is both, lucrative and a prevalent issue.

# Part III

# Defenses

# 8 | Elysium

## *Healing Vulnerable Smart Contracts via Context-Aware Patching*

*In this chapter, we propose a novel method towards automatically patching vulnerable smart contracts prior to deployment by leveraging a combination of template-based and semantic-based patching. Several approaches have been proposed to improve smart contract security, by either automatically detecting bugs prior to deployment or allowing contracts to be updated after deployment. However, merely identifying bugs automatically is not enough. This became evident when the Parity wallet was hacked a second time after being manually patched following a security audit. The most elegant solution would be to automatically eliminate bugs prior to deployment. Automatic pre-deployment patching offers a powerful promise to strengthen smart contract defenses. Current approaches are limited in the types of vulnerabilities that can be patched, in the flexibility of the patching process, and in terms of scalability. In this chapter, we propose* ELYSIUM *– a scalable approach towards automatic smart contract repair, that combines template-based with semantic-based patching by inferring context information from the bytecode.* ELYSIUM *can currently patch 7 known vulnerabilities in smart contracts automatically and it can easily be extended with new templates and new bug-finding tools. We evaluate the effectiveness and correctness of* ELYSIUM *using 3 different datasets by replaying more than 500K transactions on patched contracts, and find that* ELYSIUM *outperforms existing tools by patching at least 30% more contracts. Finally, we also compare the overhead in terms of deployment and transaction cost. In comparison to other tools,* ELYSIUM *minimizes transaction cost (up to a factor of 1.9), for only a marginally higher deployment cost.*

## 8.1 Introduction

Smart contracts may govern assets that are sometimes worth millions. This makes them an appetizing target for attackers. The underlying blockchain technology guarantees that the past cannot be changed. As a result, it is not possible to update the code of a smart contract, even if bugs are found. Moreover, blockchain technology also guarantees transparency. Thus, smart contract code is publicly visible and can be inspected by anyone for

bugs. Several high-profile attacks on high-value smart contracts have occurred. A prominent example is the 2016 DAO hack, where an attacker managed to steel ether worth 50M USD by exploiting a reentrancy bug in the smart contract [57].

Since then, various ways to mitigate the problem of insecure smart contracts have been studied. Some works have focused on adding updateability to smart contracts by using a "proxy" contract that forwards calls to the most up-to-date version of the code [225, 216]. Other works have focused on modifying clients to block transactions that might result in hacks (e.g., [69, 165, 180]). Others focused on identifying bugs prior to deployment. Many studies used symbolic execution to this end (e.g., [51, 122, 120, 102, 158, 116]), while others used abstract interpretation and model checking (e.g., [112, 136, 90, 193, 181]), including fuzzing (e.g., [110, 160, 210]). Further additions in this arsenal are tools to detect and study attacks (e.g., [175, 211]). Despite all these efforts, even well-studied bugs with well-known countermeasures (e.g., reentrancy) still occur in high-value contracts and are triggered deliberately or unwittingly. Examples include the 30M USD Parity hack in 2017 [84], a follow-up attack of its fix that resulted in locking up 150M USD [80], the 2018 reentrancy bug in the Spankchain smart contract [125] affecting 38K USD, and the 2020 reentrancy bugs in the Uniswap and Lendf.me smart contracts [191] affecting together 25M USD.

Ideally, smart contracts should be deployed as secure as possible. This entails not only proxying them, but also checking for known classes of bugs pre-deployment. Pre-deployment bug fixing typically relies on manual analysis and patching. However, manual patching is cumbersome, time consuming and, as illustrated by the second Parity hack [80], does not guarantee the absence of known classes of bugs. Automation of both bug finding and bug patching is thus needed. While there has been research into automatically patching smart contracts [199, 202, 219, 214], existing work is still limited: (1) they only address some vulnerabilities, (2) they use inflexible hard-coded templates that do not scale well, and (3) they add a large overhead in terms of deployment and transaction costs.

We propose a methodology to address these shortcomings by automatically generating context-aware patches tailored to each contract. For each contract, we perform a number of analyses such as integer type inference and free storage space inference to sufficiently understand the context of the smart contract to be able to create tailored and efficient patches. Inference and patching are performed at the bytecode level. This is obviously more challenging than working at the source code level, but has the advantage that our methodology can be applied to any smart contract, independently of the programming language that was used. An added bonus is that bytecode level patching results in more efficient code in terms of size and gas usage than recompiled patched source code [219]. Our methodology leverages a hybrid approach by combining the flexibility of template-based approaches with the effectiveness of semantic-based approaches. Users can write patching templates that contain place holders which are replaced with contract-related information during patch generation. Moreover, since our approach leverages already existing bug-finding tools, it

can easily be extended to incorporate new bug-finding tools, giving it the flexibility to handle future vulnerabilities. In summary, this chapter makes the following contributions:

**Contributions**

- We present a novel context-aware patching approach that combines template-based with semantic-based patching to create flexible and tailored patches for smart contracts.
- We propose ELYSIUM, a tool that implements our approach to automatically patch 7 different types of vulnerabilities in smart contracts at the bytecode level.
- We compare our tool to existing works using 3 different datasets and replaying more than 500K transactions, and demonstrate that ELYSIUM not only patches more bugs (at least 30% more), but that it is also more efficient in terms of gas consumption (up to 1.9 times less gas).

## 8.2 Methodology

In this section, we describe the individual challenges as well as our approach towards patching vulnerabilities at the bytecode level for the vulnerabilities listed in Table 8.1.

### 8.2.1 Smart Contract Vulnerabilities

In the last years, a plethora of smart contract vulnerabilities have been identified and studied [60, 218]. The NCC Group initiated the Decentralized Application Security Project (DASP) with the goal of grouping the most common smart contract vulnerabilities into categories and ranking them based on their real-world impact [105]. Table 8.1, lists the top 5 categories and their associated vulnerabilities. Although more categories and vulnerabilities exist, our work primarily focuses on the vulnerabilities associated with the top 5 for two reasons: 1) bytecode level detection tools exist for detecting those vulnerabilities 2) those vulnerabilities can be patched by manipulating the bytecode.

**Table 8.1:** Decentralized Application Security Project Top 5

| Rank | Category | Associated Vulnerabilities |
|---|---|---|
| 1 | Reentrancy | Same- and Cross-Function Reentrancy |
| 2 | Access Control | Transaction Origin, Suicidal, Leaking, Unsafe Delegatecall |
| 3 | Arithmetic | Integer Overflows and Underflows |
| 4 | Unchecked Low Level Calls | Unhandled Exceptions |
| 5 | Denial of Services | Unhandled Exceptions, Transaction Origin, Suicidal, Leaking, Unsafe Delegatecall |

```
1 mapping (address => uint) public
      userBalances;
2 ...
3 function withdrawBalance() public {
4   uint amount = userBalances[msg.sender];
5   msg.sender.call.value(amount)("");
6   userBalances[msg.sender] = 0;
7 }
```

**(a)** Before Patching

```
1  mapping (address => uint) public
       userBalances;
2  + bool private locked = false;
3  ...
4  function withdrawBalance() public {
5    uint amount = userBalances[msg.sender];
6  + require(!locked);
7  + locked = true;
8    msg.sender.call.value(amount)("");
9  + locked = false;
10   userBalances[msg.sender] = 0;
11 }
```

**(b)** After Patching

**Figure 8.1:** (a) Example of a function vulnerable to reentrancy due to an unguarded external call. (b) Example of a function not vulnerable to reentrancy due to a state variable guarding the external call.

### 8.2.2 Patching Reentrancy Bugs

The code snippet in Figure 8.1a provides an example of a function that is vulnerable to reentrancy at line 5. The function `withdrawBalance` transfers the balance of a user to the calling address. Note that a transfer is simply a call to an address. Hence, if `msg.sender` is a contract, then the transfer will trigger the code that is associated to `msg.sender`. This code can be malicious and call back the `withdrawBalance` function, and reenter function `withdrawBalance` while the first invocation has not finished yet. The issue here is that `userBalances[msg.sender]` has not been set to zero at that moment, and therefore an attacker can repeatedly withdraw its balance from the contract. This is clearly a concurrency issue that can be addressed in several ways. One solution, is to ensure that all state changes, such as the setting of `userBalances[msg.sender]` to zero, are performed before the call. However, this requires correctly identifying all state variable assignments that are affected by the call, and moving them before the call. Unfortunately, this process is rather tedious and error-prone, as it might break the semantics of a contract. A far more simple and less invasive approach, is to make use of *mutual exclusion*, a well studied paradigm from concurrent computing with the purpose of preventing race conditions [8]. The idea is to introduce a so-called *mutex* variable that locks the execution state and prevents concurrent access to a given resource. Figure 8.1b depicts a patched version of the function `withdrawBalance` using mutual exclusion. A new state variable called `locked` has been introduced at line 2. The variable is used as a mutex variable and is initially set to `false`. The condition at line 6 first checks if `locked` is set to `false` before executing the call at line 8. Then, before executing the call, the variable `locked` is set to `true` and when the call has finished executing, the variable is set back to `false`. This mechanism ensures that the call at line 8 is not re-executed when the function `withdrawBalance` is reentered. Nevertheless, special care needs to be taken when working

```
1 address public owner;
2 ...
3 function withdraw(address receiver)
      public {
4   require(tx.origin == owner);
5   receiver.transfer(this.balance);
6 }
```

**(a)** Before Patching

```
1 address public owner;
2 ...
3 function withdraw(address receiver)
      public {
4 - require(tx.origin == owner);
5 + require(msg.sender == owner);
6   receiver.transfer(this.balance);
7 }
```

**(b)** After Patching

**Figure 8.2:** (a) Example of a function vulnerable to transaction origin due to the use of `tx.origin`. (b) Patched example using `msg.sender` instead of `tx.origin`.

with mutexes. One has to make sure that there is no possibility for a lock to be claimed and never released, otherwise a so-called *deadlock* might occur and render the smart contract unusable. However, the greatest challenge of this approach is the introduction of a new state variable at the bytecode level. While this is straightforward when working at the source code level, it becomes more challenging when working at the bytecode level, where high level information such as state variable declarations are missing. Our idea is to use bytecode level taint analysis in order to learn about occupied storage space and infer which storage space is still available for inserting a new state variable (cf. Section 8.3 for more details on free storage space inference). It is crucial that we only introduce mutex variables at free storage space as otherwise we will overwrite already used storage space and break the semantics of the contract. Please note that the code presented in Figure 8.1a is an example of a so-called *same-function* reentrancy. However, Rodler et al. [165] presented other types of reentrancy such as *cross-function* reentrancy, *delegated* reentrancy, and *create-based* reentrancy. The idea is that an attacker can take advantage of a different function that shares the same state with the reentrancy vulnerable function. Thus, for a contract to be safe against any type of reentrancy, we have to apply the same locking mechanism to every function that shares state with the function that is vulnerable to reentrancy. We achieve this by searching the bytecode for writes to the same state variable used inside the reentrancy vulnerable function and by guarding them using the same mutex variable that is used in the reentrancy vulnerable function.

### 8.2.3 Patching Access Control Bugs

Access control bugs includes: transaction origin, suicidal, leaking, and unsafe delegatecall. While the former requires its own approach, the latter three can be patched using a common approach.

**Patching Transaction Origin.** The function `withdraw` in Figure 8.2a makes use of `tx.origin` to check if the calling address is equivalent to the owner. However, as `tx.origin` does not return the last calling address but the address that initiated the transaction, an attacker can

```
1 contract Suicidal {
2   ...
3   function kill() public {
4     selfdestruct(msg.sender);
5   }
6 }
```

**(a)** Before Patching

```
1  contract NonSuicidal {
2  + address private owner;
3      ...
4  + constructor() {
5  +   owner = msg.sender;
6  + }
7    ...
8    function kill() public {
9  +   require(msg.sender == owner);
10     selfdestruct(msg.sender);
11   }
12 }
```

**(b)** After Patching

**Figure 8.3:** (a) Example of a suicidal contract due to an unprotected `selfdestruct`. (b) Example of a non-suicidal contract due to a protected `selfdestruct`.

try to forward a transaction initiated by the owner in order to impersonate itself as the owner and bypass the check at line 4. The process of patching a transaction origin vulnerability is rather simple. Figure 8.2b depicts a patched version of the function `withdraw`. The patch simply replaces `tx.origin` with `msg.sender`, which returns the latest calling address instead of the origin.

**Patching Suicidal, Leaking, and Unsafe Delegatecall.** The contract in Figure 8.3a is considered suicidal. The function `kill` does not verify the calling address. As a result, anyone can destroy the contract. The vulnerabilities leaking and unsafe delegatecall are similar, although they relate to contracts that allow anyone to either withdraw ether or control the destination of a delegatecall. These three vulnerabilities share the same issue, namely the unprotected access to a critical operation. The idea is therefore to add the missing logic that limits the access to a critical operation to only a single entity, for example, the creator of the smart contract. Figure 8.3b depicts a patched version of the function `kill`. A new state variable `owner` has been added (line 2) as well as a constructor (lines 4-6) in order to initialize the variable `owner` during deployment with the address of the contract creator. Finally, a check has been added at line 9 to verify if `msg.sender` is equivalent to the address stored in the variable `owner`. Similar to reentrancy, this approach requires the identification of free storage space in order to introduce a new state variable `owner`. To initialize the variable `owner` at deployment (cf. Section 8.3 for more details on modifying the deployment bytecode), we are required to modify the deployment bytecode instead of the runtime bytecode. Please note that before creating a new owner variable, we first try to infer and reuse existing owner variables by employing certain heuristics (e.g., identify variables where `msg.sender` is written to). Also note that, deployment bytecode always contains a constructor at the bytecode level, we therefore just append an assignment to the end of the existing constructor bytecode.

```
1 mapping (address => uint32) public
      tokens;
2 ...
3 function buy(uint32 amount) public {
4   require(msg.value == amount);
5   tokens[msg.sender] += amount;
6 }
```

**(a)** Before Patching

```
1 mapping (address => uint32) public
      tokens;
2 ...
3 function buy(uint32 amount) public {
4   require(msg.value == amount);
5 + uint32 bounds = 2**32-1 - tokens[msg.
      sender];
6 + require(bounds >= amount);
7   tokens[msg.sender] += amount;
8 }
```

**(b)** After Patching

**Figure 8.4:** (a) Example of a function vulnerable to an integer overflow due to a missing bounds check guarding the update of `tokens[msg.sender]`. (b) Example of a function not vulnerable to integer overflows due to an added bounds check guarding the update of `tokens[msg.sender]`.

### 8.2.4 Patching Arithmetic Bugs

Arithmetic bugs such as integer overflows and underflows are a common issue in smart contracts. In 2018, several ERC-20 token smart contracts have been victims to attacks due to integer overflows [109]. The code snippet in Figure 8.4a, provides an example of a function that is vulnerable to an integer overflow at line 5. The function `buy` is missing a check that verifies if the value contained in `tokens[msg.sender]` would overflow if `amount` would be added. A common way to ensure that unsigned integer operations do not wrap, is to use the *SafeMath* library provided by OpenZeppelin [124]. For example, in the case of addition, the library performs a post-condition test, where is first computes the result of $a + b$ and then checks if the result is smaller than $a$. If this is the case, then an overflow has happened and the library halts and reverts the execution. However, Solidity allows developers to make use of smaller types (e.g., `uint32`, `uint16`, etc.) in order to use less storage space and therefore reduce costs, despite the EVM being able to operate only on 256-bit values. As a result, the Solidity compiler artificially enforces the wrapping of integers on these smaller types to be consistent with the wrapping performed by the EVM on types of 256-bit. Unfortunately, the checks provided by the SafeMath library only work with values of type `uint256` and do not protect the developers from integer overflows caused by variables of smaller types. Moreover, Solidity enables integer variables to be unsigned or signed, but SafeMath only checks for unsigned integers. Therefore, in order to be able to patch any type of integer overflow, we need to be capable of inferring the size and the signedness (i.e., signed or unsigned) of an integer variable. While this is trivial when working with source code, it becomes challenging when working with bytecode, where high-level information such as size and signedness are not directly accessible. The idea of our approach is to leverage bytecode level taint analysis in order to infer the size as well as the signedness of integer variables (cf. Section 8.3 for more details on integer type inference). Once the size

```
1 uint public prize;
2 address public winner;
3 bool public claimed = false;
4 ...
5 function claimPrize() public {
6   require(!claimed && msg.sender ==
        winner);
7   msg.sender.send(prize);
8   claimed = true;
9 }
```

**(a)** Before Patching

```
1 uint public prize;
2 address public winner;
3 bool public claimed = false;
4 ...
5 function claimPrize() public {
6   require(!claimed && msg.sender ==
        winner);
7 - msg.sender.send(prize);
8 + require(msg.sender.send(prize));
9   claimed = true;
10 }
```

**(b)** After Patching

**Figure 8.5:** (a) Example of a function vulnerable to an unhandled exception due to a missing return value check on `send`. (b) Example of a function not vulnerable to unhandled exceptions due to an added return value check on `send`.

and the signedness are determined, we can generate a patch that verifies if an arithmetic operation is in bounds with respect to size and signedness. For example, Figure 8.4b depicts a patched version of the function `buy`. First, we compute the bounds by subtracting the value of `tokens[msg.sender]` from the largest possible value of an unsigned 32-bit integer (i.e., $2^{32} - 1$) (line 5). Afterwards, we check if `amount` is smaller or equal to the computed bounds (line 6). If `amount` is not within the computed bounds, then we halt and revert the execution. Otherwise, the addition at line 7 is considered safe and we continue the execution.

### 8.2.5 Patching Unchecked Low Level Calls Bugs

An unchecked low level call, also known as an unhandled exception, occurs whenever the return value of a call is not checked. A call can fail due to several reasons: an out-of-gas exception, a revert triggered by the called contract, etc.. A developer should therefore never assume that a call is always successful, but should always check the return value and handle the case when the call fails. The function `claimPrize()` in Figure 8.5a does not check if `prize` has been rightfully sent to `msg.sender` (cf. line 7). As a result, the variable `claimed` is set to `true`, while `msg.sender` has not received the prize. Fortunately, patching an unchecked low level call is rather trivial. A patched version of the function is shown in Figure 8.5b. The patch surrounds the `send` with a `require`, which will halt the execution and revert the state in case `send` is not successful. Please note that, while this patches the unchecked low level call, the use of `require` can make in this case the contract vulnerable to denial-of-service attacks if calling `msg.sender` will always fail.

## 8.3 Design and Implementation

In this section, we provide details on the overall design and implementation of ELYSIUM.

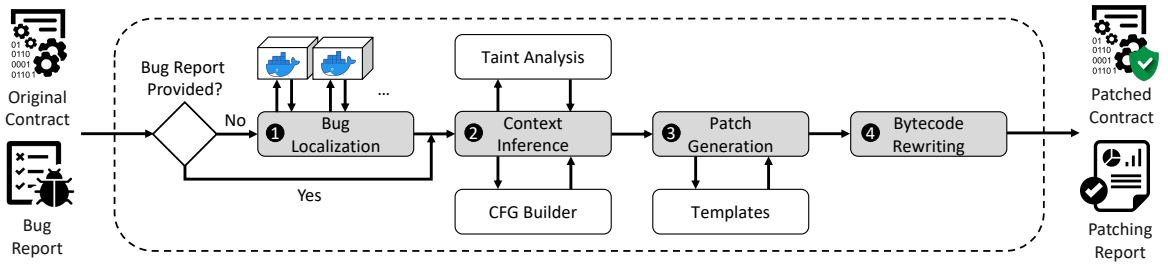



**Figure 8.6:** Overview of ELYSIUM's architecture. The shaded boxes represent the four main steps of ELYSIUM.

### 8.3.1 Overview

An overview of ELYSIUM's architecture is depicted in Figure 6.10. ELYSIUM takes as input a smart contract as well as an optional bug report and outputs a patched smart contract together with a patching report. The input smart contract can be either bytecode or Solidity source code. The latter, will be compiled into bytecode before performing any analysis or patching. The patched smart contract consists of the patched version of the bytecode of the original smart contract. The patching report contains information about execution time and the individual patches that have been applied. ELYSIUM's patching process follows four main steps: ❶ *bug localization*, ❷ *context inference*, ❸ *patch generation*, and ❹ *bytecode rewriting*. The bug localization step is responsible for detecting and localizing bugs in the bytecode. This step is skipped in case a bug report is provided. The context inference step is in charge of building the Control-Flow Graph (CFG) from the byteocde and inferring from the CFG context related information, such as integer types and free storage space, by leveraging taint analysis. The patch generation step is responsible for creating patches by inserting previously inferred context information within given patching templates. Finally, as a last step, the bytecode rewriting is in charge of injecting the generated patches into the original CFG and translating it back to bytecode. ELYSIUM is written in Python and consists of roughly 1,600 lines of code. In the following, we describe each of the four steps in more detail.

### 8.3.2 Bug Localization

In order to be able to patch bugs, ELYSIUM first needs to know the exact location of a bug and its type. One option is to implement our own bug detection solution. However, this is time consuming and error-prone. Another option is to make use of already existing bug detection solutions for smart contracts and to simply incorporate them into ELYSIUM. This approach has the advantage of adding modularity by decoupling the detection process from the patching process. This also makes it easy to extend ELYSIUM with other or future security analysis tools. ELYSIUM leverages the following three well-known smart contract analysis tools to detect and localize bugs: OSIRIS [102] to detect integer overflows, OYENTE [51] to

detect reentrancy, and MYTHRIL [120] to detect unhandled exceptions, transaction origin, suicidal contracts, leaking ether, and unsafe delegatecalls. These tools are provided to ELYSIUM as Docker images. ELYSIUM spawns each tool as a separate Docker container, and once a tool has finished running, the output of the tool is parsed and bug information such as the opcode (e.g., `CALL`, `ADD`), exact bytecode location (i.e., program counter) and vulnerability type (e.g., reentrancy, integer overflow, etc.) is extracted. This information is then added to a bug report and used by the subsequent steps. Please note that, one can also directly provide a manually crafted bug report to ELYSIUM. In such a case, ELYSIUM will skip the bug localization step and will directly forward the bug report to the subsequent steps. A user only has to ensure that the bug report follows ELYSIUM's JSON format and that it contains the aforementioned information.

### 8.3.3 Context Inference

To effectively patch vulnerabilities related to reentrancy, access control, and integer overflows, we require some context related information. We gather this information by traversing the CFG and leveraging taint analysis to infer information about integer types and free storage space. We build the CFG by using the *EVM CFG Builder* python library [213].

**Integer Type Inference.** Integer type information is composed of a size (e.g., 32-bit for type `uint32`) and a signedness (e.g., signed for type `int` and unsigned for type `uint`). Both are essential in order to correctly check whether the result of an arithmetic operation is either in-bound or out-of-bound. However, type information is usually lost during compilation and it is therefore only available at the source code level. Fortunately, we can leverage some behavioral patterns of the Solidity compiler in order to infer the size as well as the signedness of integers. For example, for unsigned integers, we know that the compiler introduces an `AND` bitmask in order to "mask off" bits that are not in-bounds with the integer's size (i.e., a zero will mask off the bit, whereas a one will leave the bit set). Thus, a variable of type `uint32` will result in the compiler adding to the bytecode a `PUSH` instruction that pushes a bitmask with the value `0xffffffff` onto the stack followed by an `AND` instruction. Hence, from the `AND` instruction we infer that it is an unsigned integer and from the bitmask we infer that its size is 32-bit, since `0xffffffff` = $2^{32} - 1$. For signed integers, the compiler will introduce a sign extension via the `SIGNEXTEND` instruction. A sign extension is the operation of increasing the number of bits of a binary number while preserving the number's sign and value. The EVM uses two's complement to represent signed integers. In two's complement, a sign extension is achieved by appending ones to the most significant side of the number. The number of ones is computed using $256 - 8(x + 1)$, where $x$ is the first value passed to `SIGNEXTEND`. For example, a variable of type `int32` will result in the compiler adding to the bytecode a `PUSH` instruction that pushes the value $3$ onto the stack followed by a `SIGNEXTEND`. Hence, from the `SIGNEXTEND` instruction we infer that it is a signed integer and from the value $3$

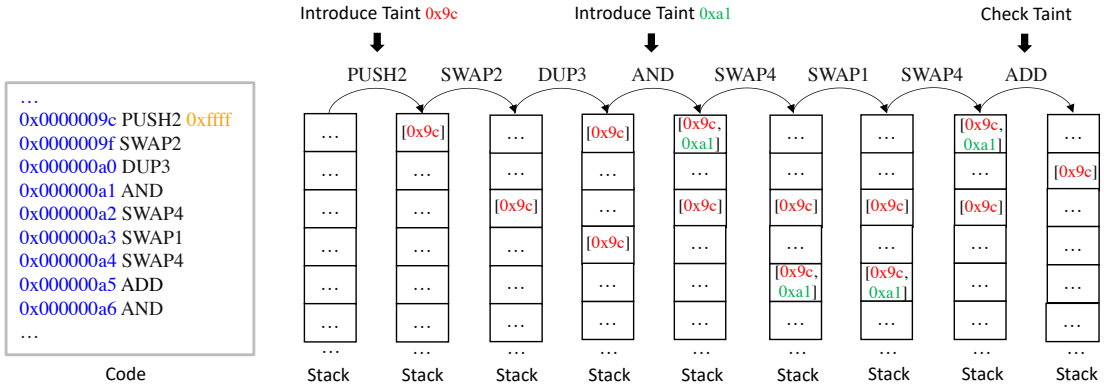



**Figure 8.7:** An example on the usage of taint analysis to infer integer types from bytecode.

we infer that its size is 32-bit, by solving the following equation: $y = 8(x + 1)$, where in this case $x = 3$. Knowing these patterns, we can use taint analysis to infer integer type information at the bytecode level. First, we iterate in a Breadth First Search (BFS) manner through the CFG until we find the basic block that contains the instruction that is labeled as the bug location. In the case of integer overflows, the instruction at the bug location can either be an `ADD`, a `SUB`, or a `MUL`. Afterwards, we use recursion to iterate from the basic block containing the bug back to the root of the CFG, thereby creating along the way a list of all visited instructions. This list of instructions reflects the execution path that has to be taken in order to reach the bug location. Using this execution path, we can apply taint analysis on it, by executing instruction by instruction and simulating in an abstract manner the effects of each instruction on a shadowed stack, memory, and storage. The idea is to introduce taint whenever we come across a `PUSH`, `AND`, or `SIGNEXTEND` instruction. Finally, when we arrive at the instruction of the bug location, we check which tainted values have been propagated up to this instruction. For example, if the tainted values that reached the bug location include a `PUSH` and an `AND` instruction, then we know that it is an unsigned integer and we know its size from the value introduced by the `PUSH` instruction. Figure 8.7 provides an illustrative example on how taint is introduced at address `0x9c` and `0xa1`, and how it is propagated throughout the stack until it reaches the vulnerable instruction `ADD` at the address `0xa6`.

**Free Storage Space Inference.** Patching reentrancy and access control bugs requires the introduction of an additional state variables at the bytecode level. State variables are associated with EVM storage, a key-value store, where both keys and values are of size 256-bit. In Solidity, statically-sized variables (e.g., everything except mappings and dynamically-sized array types) are laid out contiguously in storage starting from key zero, whereas the storage location for dynamically-sized variables is computed using a hash function. Moreover, the Solidity compiler tries to pack multiple, contiguous items that need less than 256-bit into a single storage slot. To not collude with existing statically-sized state variables, we need to

find which storage keys are already in use. To do this, we first extract all the possible execution paths from the CFG by iterating through it in a Depth First Search (DFS) manner and adding each visited instruction to a list. Each list represents one execution path contained in the CFG. An execution path is terminated whenever we come across a `STOP`, `RETURN`, `SUICIDE`, `SELFDESTRUCT`, `REVERT`, `ASSERTFAIL`, or `INVALID` instruction. Moreover, whenever we run into a `JUMPI` instruction we split the execution by creating a copy of the list of instructions visited so far and continue iterating first on one branch and then on the other branch. The EVM provides two different instructions to interact with storage: `SLOAD` and `SSTORE`. The former takes as input a storage key from the stack and pushes onto the stack the value stored at that key. The latter takes as input a storage key and a value, and stores the value at the given key. Storage keys are usually pushed onto the stack as constants. Thus, whenever a storage instruction is executed (i.e., `SLOAD` or `SSTORE`), a `PUSH` instruction will be executed before at some point in the execution with the goal of pushing the storage key onto the stack for the storage instruction to use. Our idea is therefore to run our taint analysis on all the collected execution paths and to introduce taint whenever we execute a `PUSH` instruction. The taint includes the `PUSH` instruction and will be propagated across stack as well as memory. Eventually, we will reach a storage instruction, where we then simply check the taint and infer the used storage key from the propagated `PUSH` instruction. Afterwards, we add the inferred key to the list of identified storage keys $sk$. Finally, after having analyzed all execution paths, we can compute the next available free storage key as $k = max(sk) + 1$. This approach ensures that we do not collude with existing storage keys and it preserves the contiguous layout of state variables in Ethereum smart contract.

### 8.3.4 Patch Generation

To generate patches, we use a combination of template-based and semantic patching. Table 8.2 provides an overview of all templates currently offered. A patch template is selected according to the vulnerability type that is to be patched. ELYSIUM includes templates for seven vulnerability types. Moreover, existing templates can be modified or new ones added in order to patch vulnerabilities that are not supported yet by ELYSIUM. We developed our own domain-specific language (DSL) that enables users to write their own context-aware patch templates. The structure of a patch template consists of a sequence of instructions to be deleted, a sequence of instructions to be inserted, and an insert mode and constructor flag. The insert mode determines whether the instruction sequence to be inserted should be inserted before or after the bug location. The constructor flag specifies if the deletion and insertion should occur at the deployment bytecode. Our DSL is a combination of the mnemonic representation of EVM instructions and custom keywords that act as place holders for context dependent information. We leverage the *pyevmasm* library [183] to translate the mnemonic representation of EVM instructions into EVM bytecode. The following

**Table 8.2:** Patch templates provided by ELYSIUM.

| Vulnerability | Patch Template | Source Code Representation |
|---|---|---|
| Reentrancy | {"delete": "", "insert": "free_storage_location SLOAD PUSH1_0x1 EQ ISZERO PUSH_jump_loc_1 JUMPI PUSH1_0x1 DUP1 REVERT JUMPDEST_jump_loc_1 PUSH1_0x1 free_storage_location SSTORE", "insert_mode": "before", "constructor": false}<br>...<br>{"delete": "", "insert": "PUSH1_0x0 free_storage_location SSTORE", "insert_mode": "after", "constructor": false} | + require(!locked);<br>+ locked = true;<br><br><br><br>...<br>+ locked = false; |
| Transaction Origin | {"delete": "ORIGIN", "insert": "CALLER", "insert_mode": "before", "constructor": false} | - require(tx.origin == ...);<br>+ require(msg.sender == ...); |
| Suicidal, Leaking & Unsafe Delegatecall | {"delete": "", "insert": "CALLER free_storage_location SSTORE", "insert_mode": "after", "constructor": true}<br>...<br>{"delete": "", "insert": "free_storage_location SLOAD PUSH20_0xffffffffffffffffffffffffffffffffffffff AND CALLER EQ PUSH_jump_loc_1 JUMPI PUSH1_0x1 DUP1 REVERT JUMPDEST_jump_loc_1", "insert_mode": "before", "constructor": false} | + constructor() {<br>+ owner = msg.sender;<br>+ }<br><br>...<br>+ require(msg.sender == owner); |
| Integer Overflow (Addition) | {"delete": "", "insert": "DUP2 DUP2 integer_bounds SUB LT ISZERO PUSH_jump_loc_1 JUMPI PUSH1_0x1 DUP1 REVERT JUMPDEST_jump_loc_1", "insert_mode": "before", "constructor": false} | + require(MAX_VALUE - a >= b); |
| Integer Overflow (Multiplication) | {"delete": "", "insert": "DUP2 DUP2 MUL integer_bounds AND DUP3 ISZERO DUP1 PUSH_jump_loc_1 JUMPI POP DUP3 SWAP1 DIV DUP2 EQ DUP1 JUMPDEST_jump_loc_1 SWAP1 POP PUSH_jump_loc_2 JUMPI PUSH1_0x1 DUP1 REVERT JUMPDEST_jump_loc_2", "insert_mode": "before", "constructor": false} | + require(b != 0 && a * b / b == a); |
| Integer Underflow | {"delete": "", "insert": "DUP2 DUP2 LT ISZERO PUSH_jump_loc_1 JUMPI PUSH1_0x1 DUP1 REVERT JUMPDEST_jump_loc_1", "insert_mode": "before", "constructor": false} | + require(a >= b); |
| Unhandled Exception | {"delete": "", "insert": "DUP1 ISZERO ISZERO PUSH_jump_loc_1 JUMPI PUSH1_0x1 DUP1 REVERT JUMPDEST_jump_loc_1", "insert_mode": "after", "constructor": false} | + require(...); |

four keywords exist: `free_storage_location`, `integer_bounds`, `PUSH_jump_loc_{x}`, and `JUMPDEST_jump_loc_{x}`. The `free_st orage_location` keyword is used to get the current free storage location and it is automatically replaced with a `PUSH` instruction that pushes the current free storage location onto the stack when generating the patch. The `integer_bounds` keyword is used to get the integer bounds on the instruction at the bug location and it is automatically replaced with a `PUSH` instruction that pushes the inferred integer bounds onto the stack when generating the patch. The `PUSH_jump_loc_{x}` and `JUMPDEST_jump_loc_{x}` keywords work in conjunction. They are used to mark jumps across instructions within a template. The `PUSH_jump_loc_{x}` keyword is replaced in the bytecode rewriting step with a

PUSH instruction that pushes the jump address of the JUMPDEST_jump_loc_{x} keyword. The JUMPDEST_jump_loc_{x} keyword simply acts as a marker and is afterwards replaced with a normal JUMPDEST instruction.

### 8.3.5 Bytecode Rewriting

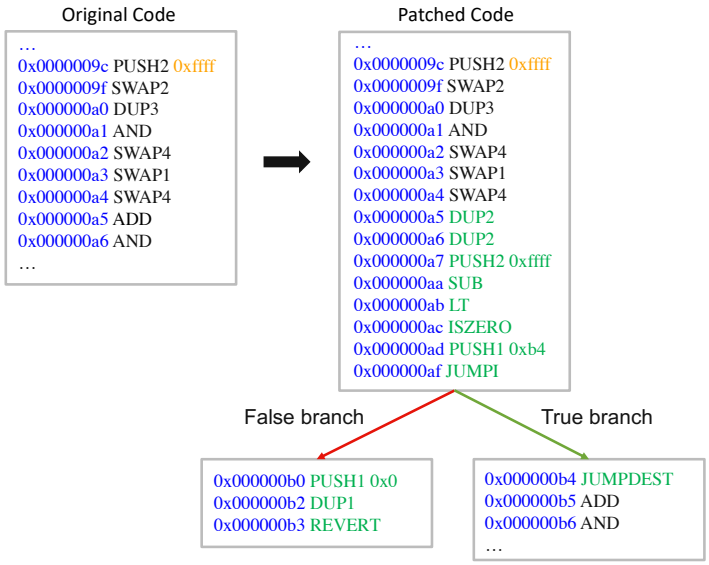



**Figure 8.8:** An example on bytecode rewriting, where a guard is added to an unguarded ADD instruction using the *integer overflow (addition)* patch template.

Ethereum smart contracts are always statically linked, meaning that the bytecode already includes all the necessary library code that is needed at runtime. This makes EVM bytecode rewriting easier than compared to traditional programs. Nonetheless, rewriting EVM bytecode still poses some challenges. Similar to traditional programs, EVM bytecode uses addresses to reference code and data in the bytecode. Thus, when modifying the bytecode, one must ensure that the addresses that reference code and data are either adjusted or preserved. There are two popular ways to deal with this issue. One solution is to preserve the layout of the existing bytecode by copying the basic block that is to be modified at the end of the bytecode. Afterwards, we replace the code of the original basic block with a jump to the copied basic block, and if needed we fill up the original basic block with useless instructions (e.g., INVALID, JUMPDEST, etc.) to preserve the original size. The modifications are then performed on the copied basic block that resides at the end of the bytecode. At the end of the modified basic block, we jump back to the end of the original basic block such that the rest of the original bytecode can be further executed. This technique is known as "trampoline" and is employed by EVMPATCH [219]. It is the least invasive, since no address references need to be adjusted. However, one disadvantage is that the original basic block needs to be large enough to at least hold the logic to jump to the end of the bytecode. Another disadvantage,

is the tremendous size increase of the bytecode. While this is less important in traditional programs, for smart contracts this has a monetary impact. The technique will add useless instructions, so-called "dead code", to preserve the layout, however, this will also result in higher deployment costs. We decided to opt for a more efficient solution in terms of deployment and transaction costs, by modifying the bytecode directly at the bug location. However, this technique requires the correct identification of broken address references and the subsequent adjustment according to the new bytecode layout. Before patching the bytecode, we create a so-called shadow address, a copy of the current address that is associated with each instruction in the CFG. Then, we scan the CFG for the basic block that is associated with the bug location. Afterwards, we modify the basic block by either deleting and/or inserting instructions according to the generated patch. Figure 8.8 depicts an example of an original basic block (left hand side) that is vulnerable to an integer overflow at address `0xa5`, and how it is patched (right hand side) by inserting a patch in the form of a guard ranging from address `0xa5` to address `0xb4`. After modifying the basic block, we update all the shadow addresses of all instructions in the CFG whose address is larger than the address of the bug location, with the size of the newly added instructions. For example, for the instruction `ADD` in Figure 8.8, we keep track of the original address with the value `0xa5` and update the shadow address to the value `0xb5` (`0xa5` + $16$ bytes of newly added instructions). After having patched all the vulnerable basic blocks, we still have to adjust the jump addresses that are pushed onto the stack since some of these might be broken (e.g., not reference to a `JUMPDEST` instruction anymore). We do this in two steps. In the first step, we localize broken jump addresses by iterating through each basic block contained in the CFG and scanning each basic block for `JUMPDEST` instructions where the original address is different than the shadow address. In the second step, we iterate through each basic block contained in the CFG and scan each basic block for `PUSH` instructions whose push value is equivalent to the original address and replace the push value with the shadow address. Finally, we convert the patched CFG back to bytecode, by first sorting the basic blocks in ascending order according to their starting, and then translating each EVM instruction within the basic block to their bytecode representation. However, remember that the deployment bytecode copies during deployment the entire runtime bytecode of the smart contract into memory. Thus, as the size of the runtime bytecode has changed, the deployment bytecode also needs to be adapted to copy the new amount of runtime bytecode. We do so by scanning the deployment bytecode for the following consecutive sequence of instructions: `PUSH DUP1 PUSH PUSH CODECOPY`. The first `PUSH` instruction determines the amount of bytes to be copied, the second `PUSH` instruction determines the offset from where the bytes should be copied, and the third `PUSH` instruction determines to which offset destination in memory the bytes should be copied. We update the deployment bytecode by replacing the value of the first `PUSH` instruction with the new size of the runtime bytecode. The second `PUSH` instruction is only updated if the deployment bytecode has also been patched (e.g., constructor code

has been added as part of a patch template).

## 8.4 Evaluation

In this section, we evaluate ELYSIUM by measuring its *effectiveness*, *correctness*, and *costs*.

### 8.4.1 Experimental Setup

**Table 8.3:** A comparison of the individual patching tools evaluated in this work.

| | | | | | Vulnerabilities | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Toolname** | **Bug Localization** | **Patching Level** | **Approach** | **Availability** | **IO** | **RE** | **UE** | **TO** | **SU** | **LE** | **UD** |
| EVMPATCH [219] | Outsourced | Bytecode | Template | Not Available | ◑ | ○ | ○ | ○ | ◐ | ◐ | ◐ |
| SMARTSHIELD [202] | Outsourced | Bytecode | Template/Semantics | On Request | ◑ | ◑ | ● | ○ | ○ | ○ | ○ |
| SCREPAIR [199] | Outsourced | Source Code | Mutation | Open Source† | ◑ | ◑ | ● | ○ | ○ | ○ | ○ |
| SGUARD [214] | Insourced | Source Code | Template/Semantics | Open Source | ◑ | ● | ○ | ● | ○ | ○ | ○ |
| **ELYSIUM** | **Outsourced** | **Bytecode** | **Template/Semantics** | **Open Source** | ● | ● | ● | ● | ● | ● | ● |

† Publicly available source code does not compile. ○ Not supported. ◑ Patching partially supported. ◐ Patch template must be specified manually. ● Fully automatic patching supported. **IO**: integer overflow, **RE**: reentrancy, **UE**: unhandled exception, **SU**: suicidal, **LE**: leaking, **UD**: unsafe delegatecall.

**Baselines.** We compare ELYSIUM to the tools listed in Table 8.3. Most tools, including ELYSIUM, have their bug localization outsourced, meaning that they leverage existing security analysis tools to detect and localize bugs. SGUARD is the only tool that leverages its own bug localization. While ELYSIUM, EVMPATCH, and SMARTSHIELD insert their patches at the bytecode level, other tools such as SCREPAIR and SGUARD insert their patches at the source code level. Almost all tools, except for SCREPAIR, use a template-based approach to introduce their patches. However, some tools such as ELYSIUM, SMARTSHIELD, and SGUARD use a combination of template-based and semantic-aware patching. The source code of EVMPATCH is not publicly available. Nonetheless, the authors released a public dataset with their results for comparison [196]. SMARTSHIELD is only available upon request. While the source code of SCREPAIR is publicly available, we did not manage to compile it. Both, ELYSIUM and SGUARD, are (will be) publicly available under an open source license. None of the aforementioned tools, except ELYSIUM, are able to patch all the vulnerabilities mentioned in this paper. For example, while SMARTSHIELD and SCREPAIR provide means to patch integer overflows, reentrancy, and unhandled exceptions, they do not provide means to patch access control related bugs such as transaction origin or unsafe delegatecall. Moreover, some tools only provide partial patching capabilities for a given type of vulnerability. For instance, all tools, except ELYSIUM, only support the patching of 256-bit unsigned integers and do not support integers of smaller size. Another example is reentrancy, where tools such as SMARTSHIELD and SCREPAIR only provide support for patching

same-function reentrancy. Furthermore, some tools such as EVMPATCH require developers to write contract specific patches for access control related bugs and therefore do not provide generic fully automatic patching. ELYSIUM on the other hand, provides complete support and fully automatic patching for all vulnerabilities.

**Table 8.4:** CVE dataset overview.

| | | | Transactions | | |
|---|---|---|---|---|---|
| **Contract** | **CVE** | **Bugs** | **Total** | **Benign** | **Attacks** |
| BEC | 2018-10299 | 1 | 409,837 | 409,836 | 1 |
| SMT | 2018-10376 | 1 | 34,164 | 34,163 | 1 |
| UET | 2018-10468 | 8 | 23,725 | 23,670 | 55 |
| SCA | 2018-10706 | 9 | 281 | 280 | 1 |
| HXG | 2018-11239 | 4 | 1,284 | 1,274 | 10 |

**Table 8.5:** SMARTBUGS dataset overview.

| | | Vulnerabilities | | |
|---|---|---|---|---|
| **Category** | **Contracts** | **Annotated** | **Detected** | **Overlap** |
| Reentrancy | 31 | 32 | 29 | **28** |
| Access Control | 18 | 19 | 12 | **12** |
| Integer Overflow | 15 | 23 | 20 | **15** |
| Unhandled Exception | 52 | 75 | 21 | **21** |
| **Total** | 116 | 149 | 82 | **76** |

**Table 8.6:** HORUS dataset overview.

| | | Transactions | | |
|---|---|---|---|---|
| **Category** | **Contracts** | **Total** | **Benign** | **Attacks** |
| Reentrancy | 46 | 11,529 | 9,021 | 2,508 |
| Access Control | 589 | 4,385 | 2,533 | 1,852 |
| – Parity Wallet Hack 1 | 585 | 4,123 | 2,509 | 1,614 |
| – Parity Wallet Hack 2 | 238 | 710 | 472 | 238 |
| Integer Overflow | 125 | 52,167 | 51,724 | 443 |
| Unhandled Exception | 1,068 | 93,268 | 90,168 | 3,100 |
| **Total Unique** | **1,823** | **160,657** | **152,845** | **7,823** |

**Datasets.** We run our experiments on three different datasets. The first dataset is the CVE dataset [196] used by Rodler et al. We chose this dataset in order to be able to compare our tool with EVMPATCH. It consists of real-world ERC-20 token contracts that were victims of integer overflow attacks. Moreover, the dataset also provides a list of attacking and benign transactions (see Table 8.4). However, the dataset is limited to integer overflows and only

contains 5 contracts. The second dataset is the SMARTBUGS dataset [220]. This dataset consists of 116 manually crafted contracts with 149 annotated vulnerabilities across 4 different vulnerabilities (see Table 8.5). While the dataset brings in a large diversity of vulnerabilities, it does not contain a list of benign or attacking transactions. The third dataset that we used is the HORUS dataset [204]. The dataset consists of 1,823 unique real-world contracts vulnerable to one of 4 different vulnerabilities, with 160,657 annotated transactions, where 152,845 transactions are benign and 7,823 transactions are attacks (see Table 8.6).

### 8.4.2 Experimental Results

**Effectiveness.** We first measure the effectiveness of ELYSIUM and the other tools on the SMARTBUGS dataset. The dataset only consists of annotated contracts and does not contain attacking nor benign transactions. We therefore first run the bug-finding tools (i.e., OSIRIS, OYENTE, and MYTHRIL) on the contracts and match the reported bugs with the annotated bugs. The overlap marks the validated ground truth (see overlap in Table 8.5). From the 149 annotated bugs, only 76 bugs are detected by the bug-finding tools. Moreover, the bug-finding tools reported 6 falsely detected bugs. Next, we patch the contracts by running each of the patching tools and rerun the bug-finding tools on the patched version returned by each patching tool, and mark a bug as successfully patched if the bug-finding tool does not report a bug anymore. Table 8.7 shows that ELYSIUM is able to patch 74 out of 76 bugs, whereas SMARTSHIELD and SGUARD can only patch 43 and 32, respectively. We see that SMARTSHIELD has issues in patching reentrancy, whereas SGUARD has issues in patching integer overflows. When considering only the bug types that all three tools have in common, then we count 22, 30, and 43 patched bugs, for SMARTSHIELD, SGUARD, and ELYSIUM, respectively. This means that ELYSIUM patches at least 30% more bugs than the other tools. To measure the effectiveness of ELYSIUM and the other tools on the CVE and HORUS datasets, we re-execute the attack transactions of each dataset, once on the original bytecode and once one the patched bytecode returned by each tool. We mark an attack as successfully blocked if the patched bytecode resulted in the transaction being reverted. Table 8.8 shows that EVMPATCH, SMARTSHIELD, and ELYSIUM successfully blocked all

**Table 8.7:** Results on running bug detection tools on patched contracts from the SMARTBUGS dataset.

| Category | SMARTSHIELD | SGUARD | ELYSIUM |
|---|---|---|---|
| Reentrancy | 7 | 28 | 28 |
| Access Control | - | 2 | 10 |
| Integer Overflow | 15 | 2 | 15 |
| Unhandled Exception | 21 | - | 21 |
| **Total** | **43** | **32** | **74** |

**Table 8.8:** Results on replayed benign and attack transactions from the CVE dataset.

| | Benign Transactions | | | Attack Transactions | | |
|---|---|---|---|---|---|---|
| **Contract** | **EVMPatch** | **SmartShield** | **Elysium** | **EVMPatch** | **SmartShield** | **Elysium** |
| BEC | 409,836 | 409,836 | 409,836 | 1 | 1 | 1 |
| SMT | 34,163 | 34,163 | 34,163 | 1 | 1 | 1 |
| UET | 17,947 | 17,947 | 17,947 | 55 | 55 | 55 |
| SCA | 280 | 280 | 280 | 1 | 1 | 1 |
| HXG | 1,248 | 1,248 | 1,248 | 10 | 10 | 10 |

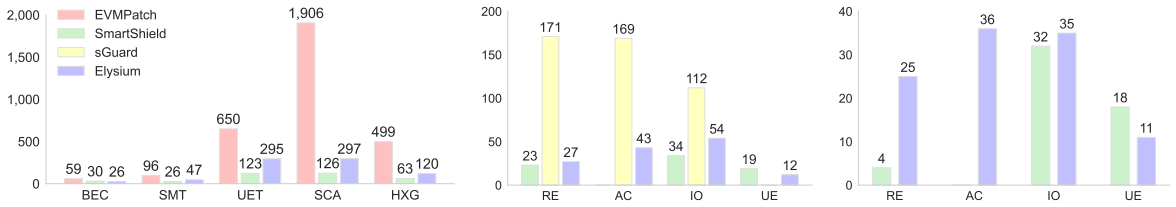
**Table 8.9:** Results on replayed benign and attack transactions from the Horus dataset.

| | Benign Transactions | | Attack Transactions | |
|---|---|---|---|---|
| **Category** | **SmartShield** | **Elysium** | **SmartShield** | **Elysium** |
| Reentrancy | 721 | 7,898 | 1,883 | 1,980 |
| Access Control | - | 2,048 | - | 1,850 |
| – Parity Wallet Hack 1 | - | 2,031 | - | 1,614 |
| – Parity Wallet Hack 2 | - | 216 | - | 236 |
| Integer Overflow | 50,394 | 45,095 | 397 | 402 |
| Unhandled Exception | 82,341 | 86,080 | 2,727 | 2,900 |
| **Total Unique** | **132,860** | **140,525** | **4,965** | **7,087** |

attacks for all the contracts within the CVE dataset. Table 8.9 shows that Elysium is able to successfully block more attacks than SmartShield on the Horus dataset.

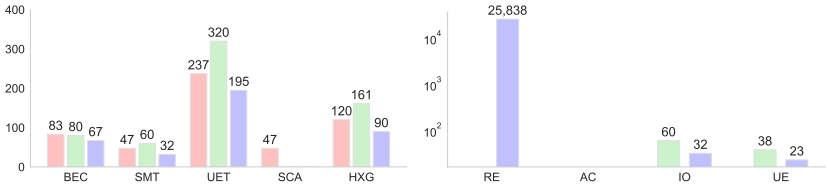
**Correctness.** Elysium's correctness depends heavily on the accurate recovery of the CFG and the accurate inference of free storage locations. We downloaded from Etherscan the bytecode and source code for the top 100 smart contracts according to their ether balance. Their lines of source code range from 19 to 3,299 and their number of functions range from 1 to 291. The *EVM CFG Builder* library [213] is able to fully recover the CFG for 85 contracts. Overall, the library achieves an average of 96% recovery with an average time of 6.7 seconds. We improved the library by adding the techniques proposed in [208]. The improved version is able to fully recover the CFG for 88 contracts and achieves an average of 98% recovery with an average time of 7.5 seconds. For the 12 non-fully recovered contracts, our improved version of the EVM CFG Builder library is able to recover on average 16% more of the CFG than the original version. To measure the accuracy of the free storage location inference employed by Elysium, we leveraged the ability of the Solidity compiler to generate the storage layout of a smart contract and compared the storage layout generated by the Solidity compiler with the storage layout inferred by Elysium. Elysium is able to correctly infer the storage layout and thus next available free storage location for all 100 contracts. Besides measuring CFG recovery and free storage location inference, we also measured

the correctness of ELYSIUM by replaying benign transactions on the patched contracts and considering them successful if the result is identical to the result of the original unpatched transaction (with the exception of the patched one using more gas). Table 8.8 shows that EVMPATCH, SMARTSHIELD, and ELYSIUM correctly executed the same number of benign transactions. However, some transactions were not counted because they resulted in an out-of-gas error due to the patched contracts consuming now more gas than originally provided to the transaction. Table 8.9 shows for the HORUS dataset, that despite ELYSIUM not being able to correctly execute as much benign transactions on integer overflow as SMARTSHIELD, overall ELYSIUM still executes more benign transactions successfully than SMARTSHIELD.




**(a)** CVE **(b)** SMARTBUGS **(c)** HORUS

**Figure 8.9:** Deployment cost increase in terms of bytes.




**(a)** CVE **(b)** HORUS

**Figure 8.10:** Transaction cost increase in terms of gas.

**Costs.** We differentiate between *deployment cost* and *transaction cost*. Deployment cost is referred to the cost when deploying a contract on the blockchain. It is computed based on the size of the bytecode. The larger the bytecode, the higher the cost. Transaction cost is referred to the cost when executing a function of a smart contract. It is computed based on the gas consumed by the executed instructions. The more expensive instructions executed, the higher the cost. While deployment cost is a one-time cost, transaction cost is a repeating cost. Our goal is therefore to primarily minimize transaction cost when introducing patches. Figure 8.9 highlights the deployment cost increase for all datasets. The deployment cost is measured by computing the difference in terms of size between the patched and the original bytecode. We state that the patches introduced by EVMPATCH and SGUARD add the largest deployment cost. This is because those tools use templates that have been generated from source code. In contrast, ELYSIUM and SMARTSHIELD, use manually crafted and optimized bytecode level templates that use less instructions. SMARTSHIELD is in most

cases the cheapest in terms of deployment cost. For example, SMARTSHIELD only adds on average 4 bytes of overhead for reentrancy on the HORUS dataset, whereas ELYSIUM adds 25 bytes. This is because SMARTSHIELD does not introduce new logic but rather tries to move writes to storage in the code. However, ELYSIUM is still by far more efficient than EVMPATCH and SGUARD. Moreover, for unhandled exceptions, ELYSIUM is more efficient than SMARTSHIELD, because of its highly optimized patch template. Figure 8.10 highlights the transaction cost increase measured for each dataset. The transaction cost is measured by computing the gas usage difference between the patched and original contract for all successfully executed benign transactions. We state that ELYSIUM adds in almost all cases the smallest overhead in terms of transaction costs. For instance, in the HORUS dataset, ELYSIUM only adds on average 32 gas units of overhead when patching integer overflows, whereas SMARTSHIELD adds 60 gas units (i.e., up to a factor of 1.9). However, we also see that ELYSIUM adds 25,838 gas units on average to patch reentrancy, whereas SMARTSHIELD adds none. This is because ELYSIUM adds two writes to storage that consume together 25,000 gas units.

## 8.5 Related Work

Framing code patching as a search and optimization problem has led several authors [17, 27] to leverage well-established heuristics and search algorithms to patch smart contacts. SCREPAIR [199] uses a genetic algorithm to find a patch. There are inherent limits in terms of quality and depth of the results. For instance, complex reentrancy patterns, such as cross-function reentrancy or faulty access control, cannot be trivially patched and contrary to claims made by SCREPAIR, patches linked to transaction order dependency are not addressed. Moreover, genetic algorithms are notoriously slow since a population of solutions needs to be evolved and this process is entirely random. Several techniques from automated program repair research have been applied to smart contracts. Nguyen et al. [214] present a tool called SGUARD, that patches smart contract vulnerabilities at the source code level. The disadvantage of this approach is that the compiler often adds unnecessary/unoptimized code, increasing bytecode size and thus causing increased deployment and transaction costs. The main difference with our work is that we patch directly at the bytecode level and can highly optimize our patches. Moreover, our tool is language independent, while SGUARD only works for Solidity. Recently, the academic community has shifted its interest to automated patching of EVM level bytecode. For instance, EVMPATCH [219] can patch integer overflow and access control patterns at bytecode level. Integer overflows are patched through hard-coded patches restricted to type `uint256` overflows and underflows. In order to patch access control patterns, the developer is required to use a custom domain-specific language for specifying a contract specific patch. Thus, patching is not fully automated anymore and the developer is required to understand and fix the bug manually. Claims that

unhandled exceptions can be patched are not backed by experiments and patching access control bugs (such as suicidal contracts and leaking contracts), is manual and tailored to the specific contract. Our approach is fully automated, covers more classes of bugs, and does not require the kind of manual preparation reported in [219]. Targeting more complex bugs, Zhang et al. [202] presented SMARTSHIELD, which automatically patches integer overflows, reentrancy bugs, and unhandled exceptions at the bytecode level. The tool is limited to only use hard-coded patches for integer overflows of type `uint256`. We observed in our experiments that SMARTSHIELD has issues in patching reentrancy bugs due the complexity of identifying data and control dependencies across bytecode. Our approach addresses these challenges by leveraging taint analysis at the bytecode level to infer contract related information (e.g., integer bounds and free storage space) and use it to generate automatically contract specific patches.

## 8.6 Conclusion

In this chapter, we proposed ELYSIUM, a tool to automatically patch smart contracts using context-related information that is inferred at the bytecode level. ELYSIUM is currently able to patch 7 types of vulnerabilities. It can easily be extended by adding further vulnerability detectors and by writing new patch templates using our custom DSL. ELYSIUM can be integrated into existing compilers and build chains to help developers automatically patch vulnerable smart contracts before deployment, independently of the programming language. We compared ELYSIUM to existing tools by patching almost 2,000 contracts and replaying more than 500K transactions. Our results show that ELYSIUM is able to effectively and correctly patch at least 30% more contracts than existing tools. Moreover, when compared to existing tools, the resulting transaction overhead is reduced by up to a factor of 1.9.

# 9 | ÆGIS

## *Shielding Vulnerable Smart Contracts Post Deployment*

*In this chapter, we study the protection of already deployed smart contracts. In recent years, smart contracts have suffered major exploits, costing millions of dollars. Unlike traditional programs, smart contracts cannot be modified once deployed. Though various tools have been proposed to detect vulnerable smart contracts, only very few solutions have been proposed so far to tackle the issue of protecting smart contracts post-deployment. The few solutions that exist often suffer from low precision and/or are not generic enough to prevent any type of attack. In this chapter, we introduce* ÆGIS*, a dynamic analysis tool that protects smart contracts from being exploited after deployment. Its capability of detecting new vulnerabilities can easily be extended through so-called attack patterns. These patterns are written in a domain-specific language that is tailored to the execution model of Ethereum smart contracts. The language enables the description of malicious control and data flows. In addition, we propose a novel mechanism to streamline and speed up the process of managing attack patterns. Patterns are voted upon and stored via a smart contract, thus leveraging the benefits of tamper-resistance and transparency provided by the blockchain. We compare* ÆGIS *to current state-of-the-art tools and demonstrate that our solution achieves higher precision in detecting attacks. Finally, we perform a large-scale analysis on the first 4.5 million blocks of the Ethereum blockchain, thereby confirming the occurrences of well reported and yet unreported attacks in the wild.*

## 9.1 Introduction

Deployed smart contracts are by default immutable, thus any bugs present during deployment [60], or as a result of changes to the blockchain protocol [144], can make a smart contract vulnerable. Moreover, since contract owners are anonymous, responsible disclosure is usually infeasible or very hard in practice. Though smart contracts can be implemented with upgradeability and destroyability in mind, this is not compulsory. As a matter of fact, Ethereum already faced several devastating attacks on vulnerable smart contracts.

In 2016, an attacker exploited a reentrancy bug in a crowdfunding smart contract known as the DAO. The attacker exploited the capability of recursively calling a payout function

contained in the contract. The attacker managed to drain over 150 million USD [57] worth of cryptocurrency from the smart contract. The DAO hack was a poignant demonstration of the impact that insecure smart contracts can have. The Ethereum market cap value dropped from over 1.6 billion USD before the attack, to values below 1 billion USD after the attack, in less than a day. Another example happened with the planned Constantinople hard fork in January 2019. Ethereum was scheduled to receive an update intended to introduce a cheaper gas cost for certain smart contract operations. On the eve of the hard fork, a new reentrancy issue caused by this update was detected. It turned out that the reduction of gas costs also enabled reentrancy attacks on smart contracts that were previously secure. This resulted in the update being delayed [144]. A third example are the Parity wallet hacks. In 2017, the Parity wallet smart contract was attacked twice due to a bug in the access control logic. The bug allowed anyone to claim ownership of the smart contract and to take control of all the funds. The first attack resulted in over 30 million USD being stolen [84], whereas the second attack resulted in roughly 155 million USD being locked forever [80].

The manner in which these issues are currently handled is not ideal. At the moment, whenever a major vulnerability is detected by the Ethereum community, it can take several days or weeks for the community to issue a critical update and even longer for all nodes to adopt this update. Such a delay extends the window for exploitation and can have dire effects on the trading value of the underlying cryptocurrency. Moreover, the lack of a standardized procedure to deal with vulnerable smart contracts, has led to a "Wild West"-like situation where several self-appointed white hats started attacking smart contracts in order to protect the funds that are at risk from other malicious attackers [142]. However, in some cases the effects of attacks can cause a split in the community so contentious that it leads to a hard fork, such as in the case of the DAO hack which led to the birth of the Ethereum classic blockchain [57].

Academia has proposed a variety of tools that allow users to scan smart contracts for vulnerabilities prior to deploying them on the blockchain or interacting with them (see e.g., [51, 116, 102, 136]). However, by design these tools cannot protect vulnerable contracts that have already been deployed. Grossman et al. [69] are the first to present ECFCHECKER, a tool that allows to dynamically check executed transactions for reentrancy. However, ECFCHECKER does not prevent reentrancy attacks. In order to protect already deployed contracts, Rodler et al. [165] propose SEREUM, a modified Ethereum client that detects and reverts[1] transactions that trigger reentrancy attacks. SEREUM leverages the principle that every exploit is performed via a transaction. Unfortunately, SEREUM has three major drawbacks. First, it requires the client to be modified whenever a new type of vulnerability is found. Second, not only the tool itself, but also any updates to it must be manually adopted by the majority of nodes for its security provisions to become effective. Third, their detection technique can only detect reentrancy attacks, despite there being many other types of

[1] Consuming gas, without letting the transaction affect the state of the blockchain.

attacks [60].

In this chapter, we propose ÆGIS, a solution that only requires a one-time modification of all the clients. Moreover, we design a new domain-specific language to describe attack patterns. These attack patterns are stored inside a smart contract, which provides a way to automatically distribute security updates to all the clients independently of the language in which they have been programmed. In summary, this chapter makes the following contributions:

**Contributions**

- We introduce a novel domain-specific language, which enables the description of so-called *attack patterns*. These patterns reflect malicious control and data flows that occur during execution of malicious transactions.
- We present ÆGIS, a tool that reverts malicious transactions in real-time using attack patterns, thereby preventing attacks on deployed smart contracts.
- We propose a novel way to quickly propagate security updates without relying on client-side update mechanisms, by making use of a smart contract to store and vote upon new attack patterns. Storing patterns in a smart contract ensures integrity, decentralizes security updates and provides full transparency on the proposed patterns.
- We illustrate the effectiveness by providing patterns to prevent the two most prominent hacks in Ethereum, the DAO and Parity wallet hacks.
- We provide a detailed comparison to current state-of-the-art runtime detection tools and perform a large-scale analysis on 4.5 million blocks. The results demonstrate that ÆGIS achieves better precision than current state-of-the-art tools.

### 9.1.1 Smart Contract Vulnerabilities

Although, a number of smart contract vulnerabilities exist [60], in this chapter, we primarily focus on two types of vulnerabilities that have been defined by the NCC Group as the top two vulnerabilities in their Decentralized Application Security Project [105]: *reentrancy* and *access control*.

**Reentrancy Vulnerabilities.** Reentrancy occurs whenever a contract calls another contract, which then calls back into the original contract, thereby creating a reentrant call. This is not an issue as long as all the state updates that depend on the call from the original contract are performed before the call. In other words, reentrancy only becomes problematic when a contract updates its state after calling another contract. A malicious contract can take advantage of this by recursively calling a contract until all the funds are drained. Figure 9.1

```
1  contract A { // Victim contract
2    ...
3    function withdraw() public {
4      if (credit[msg.sender]) {
5        msg.sender.call.value(credit[msg.sender])();
6        credit[msg.sender] = 0;
7      }
8  }
9
10 contract B { // Exploiting contract
11   ...
12   function () public payable {
13     A.withdraw();
14   }
15 }
```

**Figure 9.1:** Example of a reentrancy vulnerability.

provides an example of a malicious reentrancy. Contract $B$ contains a fallback function (line 12-14), a default function that is automatically executed when no other function is called. In this example, the fallback function of contract $B$ calls the withdraw function of contract $A$. Assuming that contract $B$ already deposited some ether in contract $A$, contract $A$ now calls contract $B$ to transfer back its deposited ether. However, the transfer results in calling the fallback function of contract $B$ once again, which results in reentering contract $A$ and once more transferring the value of the deposited ether to contract $B$. This repeats until the balance of contract $A$ becomes zero or the execution runs out of gas. Reentrancy vulnerabilities were extensively studied by Rodler et al. [165], and can be divided into four distinct categories: *same-function* reentrancy, *cross-function* reentrancy, *delegated* reentrancy and *create-based* reentrancy. Same-function reentrancy occurs whenever an attacker reenters the original contract via the same function (see Figure 9.1). Cross-function reentrancy builds on the same-function reentrancy. However, here the attacker takes advantage of another function that shares a state with the original function. Delegated reentrancy and create-based reentrancy are similar to same-function reentrancy, but use different opcodes to initiate the call. Specifically, delegated reentrancy can occur using either the `DELEGATECALL` or `CALLCODE` opcodes, while create-based reentrancy only occurs when using the `CREATE` opcode. While the `DELEGATECALL` and `CALLCODE` opcodes behave roughly similar to the `CALL` opcode, the `CREATE` opcode causes a new contract to be created and the contract constructor to be executed. This newly created contract can then call and reenter the original contract.

**Access Control Vulnerabilities.** Access control vulnerabilities result from incorrectly enforced user access control policies in smart contracts. Such vulnerabilities allow attackers to gain access to privileged contract functions that would normally not be available to them. The most famous examples of this type of vulnerability are the two Parity MultiSigWallet hacks [84, 80]. The issue originates from the fact that the developers of the Parity

```
1  contract W { // Wallet contract
2    ...
3    function W(address _owner) { // Contructor
4      L.delegatecall("initWallet(address)", _owner);
5    }
6    function () payable {
7      L.delegatecall(msg.data);
8    }
9  }
10
11 contract L { // Library contract
12   ...
13   modifier onlyOwner {
14     if (m_ownerIndex[msg.sender] > 0) _;
15   }
16   ...
17   function initWallet(address[] _owners, uint _required, uint _daylimit) {
18     initDaylimit(_daylimit);
19     initMultiowned(_owners, _required);
20   }
21   function initMultiowned(address[] _owners, uint _required) {
22     ...
23     for (uint i = 0; i < _owners.length; ++i) {
24       ...
25       m_ownerIndex[_owners[i]] = 2+i;
26     }
27     ...
28   }
29   function execute(address _to, uint _value, bytes _data) onlyOwner {
30     _to.call.value(_value)(_data));
31   }
32   function kill(address _to) onlyOwner {
33     suicide(_to);
34   }
35 }
```

**Figure 9.2:** Example of an access control vulnerability.

wallet decided to split some of the contract logic into a separate smart contract named `WalletLibrary`. This had the advantage of reusing parts of the code for multiple wallets allowing users to save on gas costs during deployment. A simplified version of the code can be seen in Figure 9.2. As can be seen in line 17-20, the initialization of the wallet is performed via the `initWallet` function located in contract $L$, which is called by the constructor of contract $W$. In addition, any unmatched function calls to contract $W$ are caught by the fallback function in line 6-8, which redirects the call to contract $L$ by means of the `DELEGATECALL` operation. Unfortunately, in the first version of the Parity MultiSig-Wallet, the developers forgot to write a safety check for the `initWallet` function, ensuring that the function can only be called once. As a result an attacker was able to gain ownership of contract $W$ by calling the `initWallet` function via the fallback function. Once in control the attacker withdrew all the funds by invoking the `execute` function (line 29-31).

After the first Parity hack, a new Parity MultiSig-Wallet Library contract was deployed ad-

dressing the issue above. In the newly deployed version, the `initWallet` function was not part of the constructor anymore, but had to be called separately after deployment. However, the developers did not call the `initWallet` function after deployment. Hence, contract $L$ remained uninitialized, meaning that the library contract itself had no owners. As a result, 3 months after deployment a user known as *devops199* was experimenting with the previous Parity hack vulnerability and called the `initWallet` function directly inside contract $L$, marking its address as the owner. The user then called the `kill` function (line 32-34), which removed the executable code of contract $L$ from the blockchain[2] and sent the remaining funds to the new owner. The contract itself contained no funds, however it was used by multiple Parity wallets which had the address of contract $L$ defined as a constant in their executable code. As a result any wallet trying to use contract $L$ as a library would now receive zero as return value, effectively rendering the wallet unusable and therefore freezing the funds contained in the wallets. This led the user to publicly disclose the steps that led to this tragedy, with the words: "I accidentally killed it." [66].

## 9.2 Methodology

In this section, we present the details of our solution towards a generic and decentralized way to prevent any type of attacks on already deployed smart contracts. Our idea is to bundle every Ethereum client with a runtime analysis tool, that interacts with the EVM and is capable of interpreting so-called *attack patterns*, and reverting transactions that match these patterns. Attack patterns are described using our domain-specific language (DSL), which is tailored to the execution model of the EVM and which allows to easily describe malicious control and data flows. The fact that we shift the capability of detecting attacks from the client-side implementation to the DSL, gives us the advantage of being able to quickly propose mitigations against new vulnerabilities, without having to modify the Ethereum client. Existing approaches, such as SEREUM for example, require the client-side implementation to be modified whenever a new vulnerability is found.

### 9.2.1 Generic Attack Detection

Attacks are detected in our system through the use of patterns, which are described using our DSL. The DSL allows for the definition of malicious events that occur during the execution of EVM instructions. The syntax of our DSL is defined by the BNF grammar in Figure 9.3. A pattern is a sequence of relations between EVM instructions that may occur at runtime. We distinguish between three types of relations, a "control flow" relation ($\Rightarrow$), a "data flow" relation ($\rightsquigarrow$), and a "follows" relation ($\rightarrow$). A control flow relation means that an instruction is control dependent on another instruction. A data-flow relation means that an instruction is

[2]The contract code is technically not removed from the blockchain, however, the contract's code can no longer be executed on the blockchain, because the contract has been marked as killed.

⟨*instr*⟩ ::= `CALL` | `CALLDATALOAD` | `SSTORE` | `JUMPI` | ...

⟨*exec*⟩ ::= **depth** | **pc** | **address** | **stack(**int**)** | **stack.result** |
| **memory(**int, int**)** | **transaction.**⟨*trans*⟩
| **block.**⟨*block*⟩

⟨*trans*⟩ ::= **hash** | **value** | **from** | **to** | ...

⟨*block*⟩ ::= **number** | **gasUsed** | **gasLimit** | ...

⟨*comp*⟩ ::= < | > | ≤ | ≥ | = | ≠ | + | - | · | /

⟨*expr*⟩ ::= **(src.**⟨*exec*⟩ ⟨*comp*⟩ ⟨*expr*⟩**)** [∧ ⟨*expr*⟩]
| **(**⟨*expr*⟩ ⟨*comp*⟩ **dst.**⟨*exec*⟩**)** [∧ ⟨*expr*⟩]
| **(src.**⟨*exec*⟩ ⟨*comp*⟩ **src.**⟨*exec*⟩**)** [∧ ⟨*expr*⟩]
| **(src.**⟨*exec*⟩ ⟨*comp*⟩ **dst.**⟨*exec*⟩**)** [∧ ⟨*expr*⟩]
| **(dst.**⟨*exec*⟩ ⟨*comp*⟩ **dst.**⟨*exec*⟩**)** [∧ ⟨*expr*⟩]
| **(src.**⟨*exec*⟩ ⟨*comp*⟩ int**)** | **(dst.**⟨*exec*⟩ ⟨*comp*⟩ int**)**

⟨*rel*⟩ ::= ⇒ | ⇝ | →

⟨*pattern*⟩ ::= **(opcode** = ⟨*instr*⟩**)** ⟨*rel*⟩ **(opcode** = ⟨*instr*⟩**)** [**where** ⟨*expr*⟩]
| ⟨*pattern*⟩ ⟨*rel*⟩ **(opcode** = ⟨*instr*⟩**)** [**where** ⟨*expr*⟩]
| **(opcode** = ⟨*instr*⟩**)** ⟨*rel*⟩ ⟨*pattern*⟩ [**where** ⟨*expr*⟩]

**Figure 9.3:** DSL for describing attack patterns.

data dependent on another instruction. A follows relation means that an instruction is executed after another instruction, without necessarily being control or data dependent on the other instruction. A relation is always between two EVM opcodes: a source opcode (`src`) and a destination opcode (`dst`). The source marks the beginning of the relation, whereas the destination defines the end of the relation. Moreover, the DSL allows to create conjunctions of expressions that allow to compare the execution environment between source and destination. The execution environment includes the current depth of the call stack (`depth`), the current value of the program counter (`pc`), the address of the contract that is currently being executed (`address`), the current values on the stack (`stack`) as well as the result of an operation that is pushed onto the stack (`stack.result`), the current values stored in memory (`memory`), and finally, properties of the current transaction that is being executed (e.g., `hash`) as well as properties of the current block that is being executed (e.g., `number`). The stack is addressable via an integer, where 0 defines the top element on the stack. The memory is addressable via two integers: an offset and a size. In the following, we explain the semantics of our DSL via two concrete examples of attack patterns: *same-function reentrancy* and the *parity wallet hack 1*.

**Same-Function Reentrancy.** Reconsider the reentrancy example that was described in

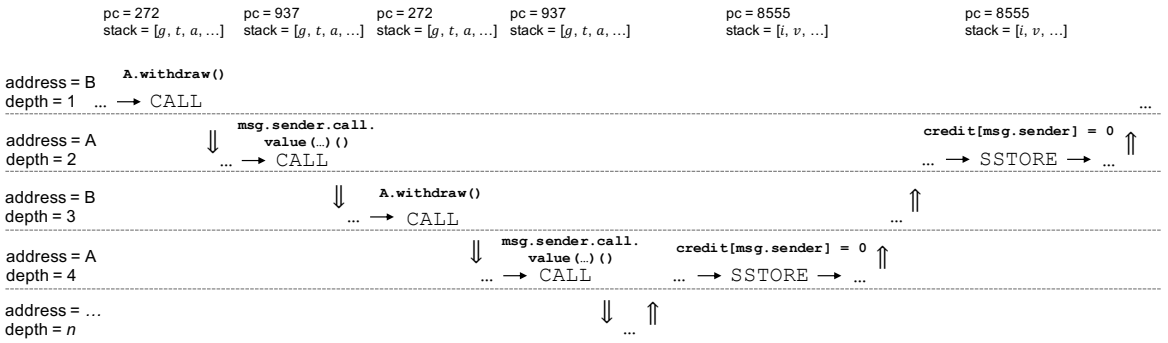



**Figure 9.4:** Execution example of a reentrancy attack, where the stack values $g$ (gas), $t$ (to), $a$ (amount), $i$ (index) and $v$ (value) represent the respective parameters passed to the instructions during execution. A control flow relation is depicted using $\Rightarrow$, while $\rightarrow$ depicts a follows relation.

Section 9.1.1. Figure 9.4, illustrates the control flow as well as the follows relations that occur during the execution of that example. The execution starts with contract address $B$ and a call stack depth of 1. Eventually, contract $B$ calls the withdraw function of contract $A$, which results in executing the `CALL` instruction and increasing the depth of the call stack to 2, and switching the address of the contract that is being executed to contract $A$. Next, contract $A$ sends some funds to contract $B$, which also results in executing the `CALL` instruction and increasing the depth of the call stack to 3, and switching the address of the contract that is being executed back to contract $B$. As a result, the fallback function of contract $B$ is called, which in turn calls again the withdraw function of contract $A$. This sequence of calls repeats until the balance of contract $A$ is either empty or the execution runs out of gas. Eventually, the state in contract $A$ is updated by executing the `SSTORE` instruction. Given these observations, we can now create the following attack pattern in order to detect and thereby prevent same-function reentrancy:

```
(opcode = CALL) ⇒ (opcode = CALL) where
 (src.stack(1) = dst.stack(1)) ∧
 (src.address = dst.address) ∧
 (src.pc = dst.pc) →
(opcode = SSTORE) → (opcode = SSTORE) where
 (src.stack(0) = dst.stack(0)) ∧
 (src.address = dst.address) ∧
 (src.depth > dst.depth)
```

This attack pattern evaluates to true if a transaction meets the following two conditions:

(1) there is a control flow relation between two `CALL` instructions, where both instructions share the same call destination (i.e., `src.stack(1) = dst.stack(1)`), are executed by the same contract (i.e., `src.address = dst.address`) and share the same program

counter (i.e., `src.pc = dst.pc`);

(2) two `SSTORE` instructions follow the previous control flow relation, where both instructions write to the same storage location (i.e., `src.stack(0) = dst.stack(0)`), are executed by the same contract (i.e., `src.address = dst.address`) and where the first instruction has a higher call stack depth than the second instruction (i.e., `src.depth > dst.depth`).

It is worth mentioning that we compare the program counter values of the two `CALL` instructions in order to make sure that it is the same function that is being called, as our goal is to detect only same-function reentrancy.

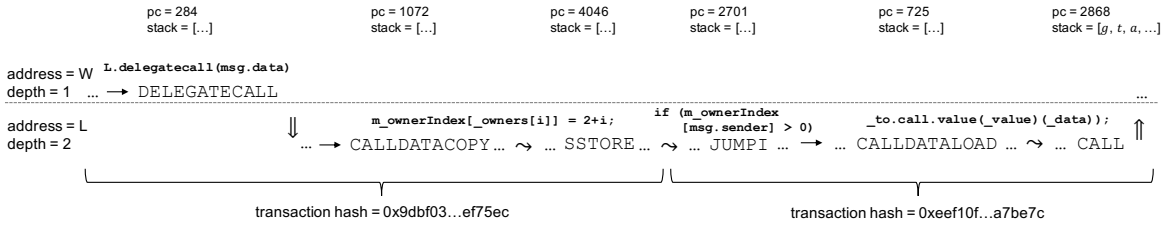



**Figure 9.5:** Execution example of an attack on an access control vulnerability. A data flow relation is depicted with $\rightsquigarrow$. The variables *g*, *t* and *a* are as discussed in Figure 9.4.

**Parity Wallet Hack 1.** Reconsider the access control example described in Section 9.1.1. Figure 9.5 illustrates the relevant control flow, data flow and follows relations that occur during the execution of that example. We note that the execution example is divided into two separate transactions. In the first transaction, the attacker sets itself as the owner, whereas in the second transaction the attacker transfers all the funds to itself. Although in reality an attacker performs two separate transactions, in our methodology, the two transactions are represented as a single sequence of execution events. For both transactions, the execution starts with contract address $W$ eventually making a delegate call to contract address $L$, as part of the attacker calling the fallback function of contract $W$. In the first transaction, we see that at a certain point contract $L$ copies data from the transaction using the `CALLDATACOPY` instruction and stores it into storage via the `SSTORE` instruction. An interesting observation here is that state is shared across transactions through storage. In the second transaction, the data that has previously been stored is now loaded onto the stack and used by a comparison. A comparison is ultimately reflected via the `JUMPI` instruction. Finally, we see that the comparison follows a `CALLDATALOAD` instruction whose data is used by a call `CALL` instruction. Given these observations, we are now able to create the following attack pattern in order to detect and thereby prevent the first Parity wallet hack:

```
(opcode = DELEGATECALL) ⇒ (opcode = CALLDATACOPY) ⇝
(opcode = SSTORE) ⇝ (opcode = JUMPI) where
  (src.transaction.hash ≠ dst.transaction.hash) →
((opcode = CALLDATALOAD) ⇝ (opcode = CALL)) where
  (dst.stack(2) > 0)
```

The above attack pattern evaluates to true if the following two conditions are met:

(1) there is a transaction with a control flow relation between a `DELEGATECALL` instruction and a `CALLDATACOPY` instruction, where the data of the `CALLDATACOPY` instruction flows into an `SSTORE` instruction;

(2) there is another transaction (i.e., `src.transaction.hash` $\neq$ `dst.transaction.hash`) where the data that has been previously stored via the `SSTORE` instruction flows into a `JUMPI` instruction and is followed by a `CALLDATALOAD` instruction whose data flows into a `CALL` instruction that sends out funds (i.e., `dst.stack(2) > 0`).

It is worth noting that the Parity wallet attack is a multi-transactional attack and that it is therefore significantly different from a reentrancy attack, that is solely based on a single transaction. For more examples of attack patterns, please refer to Table 9.1.

### 9.2.2 Decentralized Security Updates

While our approach of using a DSL allows us to have a generic solution for detecting attacks, it still leaves two open questions:

(1) How do we distribute and enforce the same patterns across all the clients?

(2) How do we decentralize the governance of patterns in order to prevent a single entity from deciding which patterns are added or removed?

The answer to these questions is to use a smart contract that is deployed on the blockchain itself. This solves the problem of distributing and enforcing that the same patterns are always used across all clients. Specifically, patterns are stored inside the smart contract and the blockchain protocol itself guarantees that every client knows about the exact same state and therefore has access to exactly the same patterns. The second problem of decentralizing the governance of patterns, is solved by allowing the proposal and voting of patterns via the smart contract as depicted in Figure 9.6. The contract maintains a list of eligible voters that vote for either accepting or rejecting a new pattern. If the majority has voted with “yes”, i.e., to accept the pattern, then it is added to the list of active patterns. In that case, every client is automatically notified through the mechanism of smart contract events, and retrieves the updated list of patterns from the blockchain. In other words, if a pattern is accepted by the voting mechanism, it is updated across all the clients through the existing

**Table 9.1:** List of vulnerabilities and their respective attack patterns.

| Vulnerability | Attack Pattern |
|---|---|
| Same-Function Reentrancy | `(opcode = CALL) ⇒ (opcode = CALL) where`<br>`  (src.stack(1) = dst.stack(1)) ∧`<br>`  (src.address = dst.address) ∧ (src.pc = dst.pc) →`<br>`(opcode = SSTORE) → (opcode = SSTORE) where`<br>`  (src.stack(0) = dst.stack(0)) ∧`<br>`  (src.address = dst.address) ∧ (src.depth > dst.depth)` |
| Cross-Function Reentrancy | `(opcode = CALL) ⇒ (opcode = CALL) where`<br>`  (src.stack(1) = dst.stack(1)) ∧ (src.address = dst.address) ∧`<br>`  (src.memory(src.stack(3), src.stack(4)) ≠`<br>`  dst.memory(dst.stack(3), dst.stack(4))) →`<br>`(opcode = SSTORE) → (opcode = SSTORE) where`<br>`  (src.stack(0) = dst.stack(0)) ∧`<br>`  (src.address = dst.address) ∧ (src.depth > dst.depth)` |
| Delegated Reentrancy | `(opcode = DELEGATECALL) ⇒ (opcode = DELEGATECALL) where`<br>`  (src.stack(1) = dst.stack(1)) ∧`<br>`  (src.address = dst.address) ∧ (src.pc = dst.pc) →`<br>`(opcode = SSTORE) → (opcode = SSTORE) where`<br>`  (src.stack(0) = dst.stack(0)) ∧`<br>`  (src.address = dst.address) ∧ (src.depth > dst.depth)` |
| | `(opcode = CALLCODE) ⇒ (opcode = CALLCODE) where`<br>`  (src.stack(1) = dst.stack(1)) ∧`<br>`  (src.address = dst.address) ∧ (src.pc = dst.pc) →`<br>`(opcode = SSTORE) → (opcode = SSTORE) where`<br>`  (src.stack(0) = dst.stack(0)) ∧`<br>`  (src.address = dst.address) ∧ (src.depth > dst.depth)` |
| Create-Based Reentrancy | `(opcode = CREATE) ⇒ (opcode = CREATE) where`<br>`  (src.stack(1) = dst.stack(1)) ∧`<br>`  (src.address = dst.address) ∧ (src.pc = dst.pc) →`<br>`(opcode = SSTORE) → (opcode = SSTORE) where`<br>`  (src.stack(0) = dst.stack(0)) ∧`<br>`  (src.address = dst.address) ∧ (src.depth > dst.depth)` |
| Parity Wallet Hack 1 | `(opcode = DELEGATECALL) ⇒ (opcode = CALLDATACOPY) ⇝`<br>`  (opcode = SSTORE) ⇝ (opcode = JUMPI) where`<br>`  (src.transaction.hash ≠ dst.transaction.hash) →`<br>`((opcode = CALLDATALOAD) ⇝ (opcode = CALL)) where`<br>`  (dst.stack(2) > 0)` |
| Parity Wallet Hack 2 | `(opcode = CALLDATACOPY) ⇝ (opcode = SSTORE) ⇝ (opcode = JUMPI) where`<br>`  (src.transaction.hash ≠ dst.transaction.hash) →`<br>`((opcode = CALLDATALOAD) ⇝ (opcode = SELFDESTRUCT))` |
| Integer Overflow (Addition) | `(opcode = CALLDATALOAD) ⇝ (opcode = ADD) where`<br>`  ((dst.stack(0) + dst.stack(1)) ≠ dst.stack.result) ⇝ (opcode = CALL)` |
| Integer Overflow (Multiplication) | `(opcode = CALLDATALOAD) ⇝ (opcode = MUL) where`<br>`  ((dst.stack(0) * dst.stack(1)) ≠ dst.stack.result) ⇝ (opcode = CALL)` |
| Integer Underflow | `(opcode = CALLDATALOAD) ⇝ (opcode = SUB) where`<br>`  ((dst.stack(0) - dst.stack(1)) ≠ dst.stack.result) ⇝ (opcode = CALL)` |
| Timestamp Dependence | `(opcode = TIMESTAMP) ⇝ (opcode = JUMPI) → (opcode = CALL) where`<br>`  (dst.stack(2) > 0)` |
| Transaction Order Dependency | `(opcode = SSTORE) ⇝ (opcode = SLOAD) where`<br>`  (src.block.number = dst.block.number) ∧`<br>`  (src.transaction.from ≠ dst.transaction.from)` |

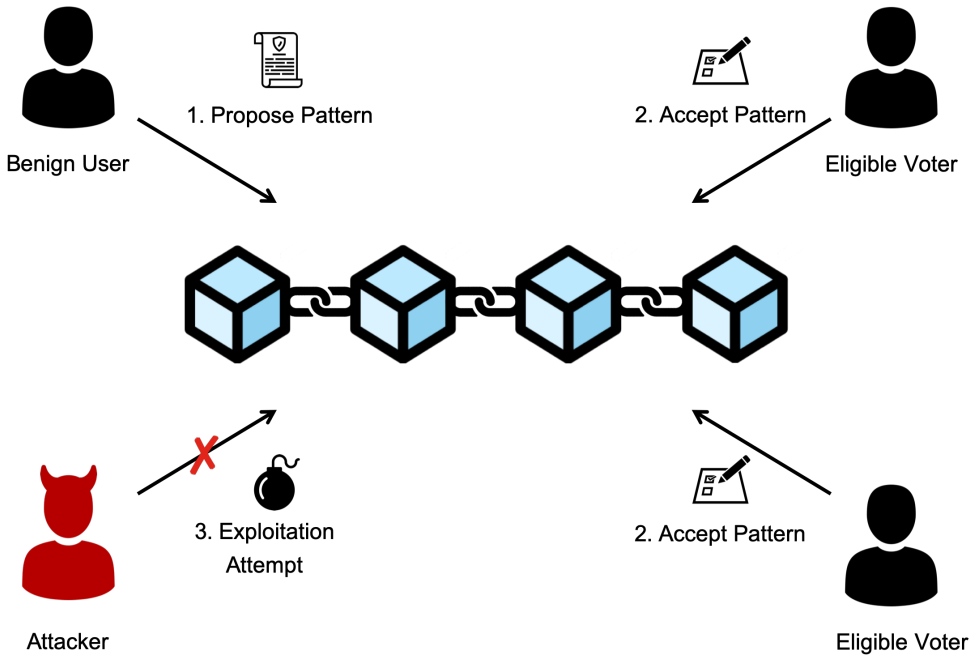



**Figure 9.6:** An illustrative example of ÆGIS's workflow: Step 1) A benign user detects a vulnerability and proposes a pattern (written using our DSL) to the smart contract. Step 2) Eligible voters vote to either accept or reject the pattern. If the majority votes to accept the pattern, then all the clients are updated and the pattern is activated. Step 3) An attacker tries but fails to exploit a vulnerable smart contract due to the voted pattern matching the malicious transaction.

consensus mechanism of the Ethereum blockchain. However, solving the second problem using a voting mechanism opens up a new problem concerning the requirements needed for governing the votes. In voting literature, verifiability and privacy are typically seen as key requirements. *Verifiability* concerns linking the output to the input in a verifiable way. *Privacy* concerns whether a vote can be linked back to a voter. In addition, we argue that the situation here is more akin to boardroom voting than to general elections, because it should be possible to hold voters *accountable*. This means that privacy must be maintained only until the election is over. Finally, the voting system must not be favorable to any voters – e.g., it should not confer an advantage to voters that cast their vote late. This final property is called *fairness*. It is worth noting that fairness requires privacy during the voting phase. This leads to the following three requirements:

(1) ***Verifiability:*** The outcome of the vote must be verifiably related to the votes as cast by the voters;

(2) ***Accountability:*** Voters can be held accountable for how they voted;

(3) ***Fairness:*** No intermediate information must be leaked.

## 9.3 ÆGIS

In this section, we provide the implementation details of our solution called ÆGIS[3]. Figure 9.7, provides an overview of the architecture of ÆGIS and highlights its main components. ÆGIS is implemented on top of Trinity[4], an Ethereum client implemented in Python.

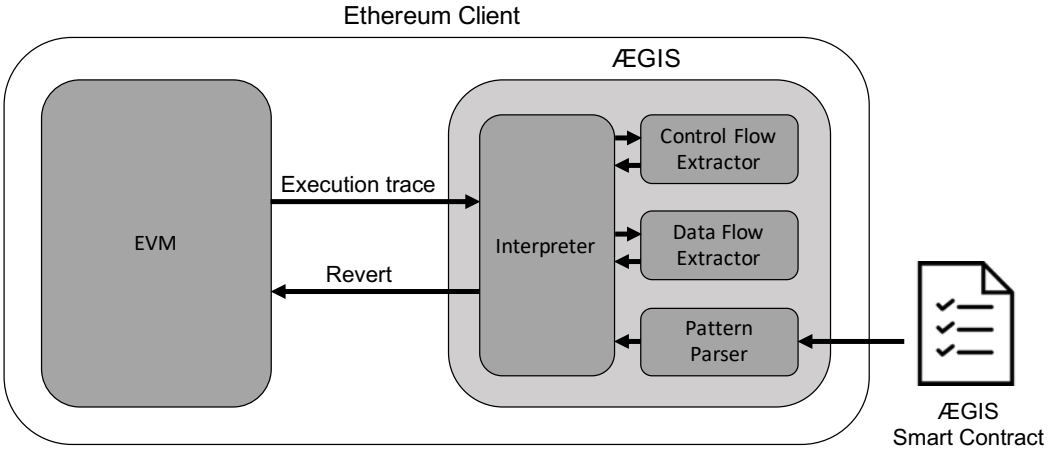



**Figure 9.7:** Architecture of ÆGIS. The dark gray boxes represent ÆGIS's main components.

### 9.3.1 Ethereum Client

**EVM.** We modified the EVM of Trinity such that it keeps track of all the executed instructions and their states at runtime, in the form of an ordered list. We refer to this list as the execution trace. Each record in this list contains the executed opcode, the value of the program counter, the depth of the call stack, the address of the contract that is being executed, and finally, all the values that were stored on the stack and in memory. This list is passed to the interpreter component of ÆGIS.

**Interpreter.** The interpreter loops through the list of executed instructions and passes the relevant instructions to the control flow and data flow extractor components. It is also responsible for signaling the EVM a revert in case the execution trace matches an attack pattern.

**Control Flow Extractor.** The control flow extractor is responsible for inferring control flow information. We do so by dynamically building a call tree from the instructions received by the interpreter. A control flow relation is reported if there exists a path along the call tree, from the source instruction to the destination instruction defined in a given pattern. Thus, control flow relations represent call dependencies between two instructions.

[3]Code is publicly available at: https://github.com/christoftorres/Aegis
[4]`https://trinity.ethereum.org/`

**Data Flow Extractor.** The data flow extractor is responsible for collecting data flow information. We track the flow of data between instructions by using dynamic taint analysis. Taint is introduced whenever we come across a source instruction and checked whenever we come across a destination instruction. Source and destination instructions are defined by a given pattern. Taint propagation follows the semantics of the EVM [30] across stack, memory and storage. We perform byte-level precision tainting. Taint that is stored across stack and memory is volatile, meaning that it is cleared across transactions. Taint that is stored across storage is persistent, meaning that it remains in storage across transactions. This allows us to perform inter-transactional taint analysis. A data flow relation is given if taint flows from a source instruction into a destination instruction.

**Pattern Parser.** The pattern parser is responsible for extracting and parsing the patterns from the voting smart contract. We implemented our pattern language using `textX`[5], a Python framework providing a meta-language for building DSLs.

### 9.3.2 ÆGIS Smart Contract

The ÆGIS smart contract ensures proper curation of the list of active patterns. We implemented our smart contract in Solidity. As previously mentioned, patterns are accepted or removed via a voting mechanism. The contract holds all proposed additions and removals of patterns and allows a vote on them within a set time window. The duration can be configured and updated by the contract owner. Proposals are open to the public and anyone can propose an addition to or removal from the list of patterns.

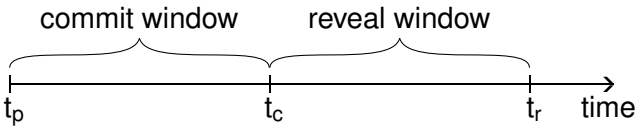



**Figure 9.8:** Timeline of the two voting stages.

**Fairness.** Votes should remain secret until all eligible voters have had sufficient opportunity to vote. Therefore, two time windows are employed. The first window is for sending a commitment that includes a deposit. The second window is for revealing a vote including the return of the committed deposits. The two windows are illustrated in Figure 9.8. In the figure, $t_p$ represents the point in time when a pattern is proposed and marks the start of the commit window. $t_c$ marks the end of the commit window and the start of the reveal window. Lastly, $t_r$ marks the end of the reveal window and the time when the pattern list is updated in case of a positive vote outcome. A commitment is a hash of the vote ID, the voter's vote and a nonce. The vote ID is a hash of the proposed pattern and identifies the pattern that is being voted

[5] `https://github.com/textX/textX`

on. The voter's vote is encoded as a Boolean. The nonce ensures that commitments cannot be replayed. The smart contract records these commitments, which must be sent with the predefined deposit and within the predefined time window. During the commitment phase no one knows how anyone else has voted on a given pattern, and so cannot be swayed by the decisions of others. However, the process should ultimately be transparent to both voters and non-voters to foster trust in the system. As such, during the second window, the reveal window, all voters reveal how they have voted. They must reveal their vote in order to get their deposit back. No commits may be made once the reveal period has started.

**Tallying.** The voting ends either when more than 50% (50% + 1 vote) of the total number of votes reaches either accept or reject, or when the time window for revealing expires with less than 50% having been reached. In case the voting has ended but the reveal window has not yet passed, any remaining voters are still eligible to reveal their vote, such that their deposit can be returned. The reveal period is bounded so that patterns are accepted or rejected in a practical amount of time. In the event of a successful vote, the pattern to which the vote pertains is added to or removed from the record held by the contract, according to the proposal. If a vote is unsuccessful, i.e., no majority voted for the proposal, the record of patterns is not changed.

**Actors.** There are three types of actors: the proposers that submit proposals to add or remove patterns, the voters that vote on proposals, and the admins that govern the list of eligible voters as well as the parameters of the smart contract (e.g., deposit, commit and reveal windows, etc.). The ÆGIS smart contract allows every user on the blockchain to become a proposer by submitting a proposal. Voters then vote on the proposals by first committing their vote and at a later stage revealing it. Not every user is an eligible voter. Voters are only those users whose account address is stored in the list of eligible voters maintained by the smart contract. Admins may update the list of eligible voters. They oversee the proper curation of the smart contract and act as a governing body. Admins are agreed upon off-chain and are represented by a multi-signature wallet. A multi-signature wallet is an account address which only performs actions if a group of users give their consent in form of a signature.

**Data Structures.** The smart contract consists of several functions and data structures that allow for the voting process to take place. We make use of a number of *modifiers*, which act as checks carried out before specific functions are executed. We use these to check that: 1) a voter is eligible, 2) a vote is in progress, 3) a reveal is in progress and 4) the associated vote has ended. We use a struct to hold the details of each vote, these include the `patternID`, the proposed `pattern` and the `startBlock`. These values enable us to record the details needed to check when a vote ends, check that the same pattern has not already been proposed, and count the number of votes. The struct is used in conjunction with a mapping, which maps a

32 bytes value to the details of each vote. The 32 bytes value represents the `voteID` of each vote, created by hashing unique vote information. A constructor is used to define, at contract launch, the value of the necessary deposit and the time windows during which voters can commit or reveal. The former is given in ether, while the latter are given in number of blocks. The deposit is used to ensure that those who committed a vote also reveal their vote. These values can be changed later using the contract's admin functions.

**Functionality.** The public functions for the voting process are: `addProposal`, `removeProposal`, `commitToVote` and `revealVote`. Both proposal functions first check if a vote with the same ID already exists, and if not create a new instance of voting details via the mapping. Next, the `commitToVote` function can be used inside the defined number of blocks to submit a unique hash of an eligible voter's vote. This function makes use of the `canVote` modifier to protect access. The voter's commitment and vote hash are stored only if the correct deposit amount was sent to the function. Once the vote stage has ended the reveal stage begins. During this window the `revealVote` function, protected by the `canVote` modifier, processes vote revelations and returns deposits. The function checks that the stored hash matches the hash calculated from the parameters passed to it, and if so, returns the voter's deposit and records the vote. Lastly, it calls an internal function which tallies the votes and adds or removes the pattern if either the for or against vote has reached over 50%. In this way the vote is self tallying. The patterns are ultimately stored in an array that can be iterated over to ensure each node has the full set. Finally, the contract also has two admin functions: `transferOwnership`, `changeVotingWindows`. Both of these are protected by the `isOwner` modifier. The former allows the current owning address to transfer control of the contract to a new address. The latter allows the commit and reveal windows to be changed as well as the amount required as a voting deposit.

## 9.4 Evaluation

In this section, we evaluate the effectiveness and correctness of ÆGIS, by conducting two experiments. In the first experiment we compare the effectiveness of ÆGIS to two state-of-the-art reentrancy detection tools: ECFCHECKER [69] and SEREUM [165]. In the second experiment we perform a large-scale analysis and measure the correctness as well as the performance of ÆGIS across the first 4.5 million blocks of the Ethereum blockchain.

### 9.4.1 Comparison to Reentrancy Detection Tools

By analyzing transactions sent to contracts, Rodler et al.'s tool SEREUM flagged 16 contracts as victims of reentrancy attacks. However, after manual investigation the authors found that only 2 out of the 16 contracts have actually become victims to reentrancy attacks. We decided to analyze these 16 contracts and see if we face the same challenges in classifying

**Table 9.2:** Comparison between SEREUM and ÆGIS on the effectiveness of detecting reentrancy attacks.

| | CCRB | DAO | 0x7484a1 | proxyCC | DAC | DSEthToken | 0x695d73 | EZC | 0x98D8A6 | WEI | 0xbD7CeC | 0xF4ee93 | Alarm | 0x771500 | KissBTC | LotteryGameLogic |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SEREUM | FP | TP | FP | FP | FP | TP | FP | FP | FP | FP | FP | FP | FP | FP | FP | FP |
| ÆGIS | TN | TP | TN | TN | TN | TP | TN | TN | TN | TN | TN | TN | TN | TN | TN | TN |

these contracts correctly. We contacted the authors of SEREUM and obtained the list of contract addresses. Afterwards, we ran ÆGIS on all transactions related to the contract addresses, up to block number 4,500,000[6]. Table 9.2 summarizes our results and provides a comparison to the results obtained by SEREUM. From Table 9.2, we can observe that ÆGIS successfully detects transactions related to the DAO contract and the DSEthToken contract, as reentrancy attacks. Moreover, ÆGIS correctly flags the remaining 14 contracts as not vulnerable. Hence, in contrast to SEREUM, ÆGIS produces no false positives on these 16 contracts. After analyzing the false positives produced by SEREUM, we conclude that ÆGIS does not produce the same false positives because first, ÆGIS does not use taint analysis in its pattern and therefore does not face issues of over-tainting, and secondly, it does not make use of dynamic write locks to detect reentrancy.

**Reentrancy with Locks.** Besides evaluating SEREUM on the set of 16 real-world smart contracts, Rodler et al. also compared SEREUM to ECFCHECKER, using self-crafted smart contracts as a benchmark [166]. The goal of this benchmark is to provide means to investigate the quality of reentrancy detection tools. The benchmark consists of three functionally equivalent contracts, except that the first contract does not employ any locking mechanism to guard the reentry of functions (*VulnBankNoLock*), the second contract employs partial implementation of a locking mechanism (*VulnBankBuggyLock*), and the third contract employs a full implementation of a locking mechanism (*VulnBankSecureLock*). As a result, the first contract is vulnerable to same-function reentrancy as well as cross-function reentrancy. The second contract is vulnerable to cross-function reentrancy, but not to same-function reentrancy. Finally, the third contract is safe regarding both types of reentrancy. We deployed these three contracts on the Ethereum test network called Ropsten and ran the three contracts against ÆGIS. Table 9.3 contains our results and compares ÆGIS to ECFCHECKER and SEREUM. We can see that ECFCHECKER has difficulties in detecting cross-function reentrancy, whereas SEREUM has difficulties in distinguishing between reentrancy and manually introduced locks. This is probably due to the locking mechanism exhibiting exactly

[6]This is the maximum block number analyzed by the authors of SEREUM.

**Table 9.3:** Comparison between ECFCHECKER, SEREUM and ÆGIS on the effectiveness of detecting same-function and cross-function reentrancy attacks with manually introduced locks.

| Smart Contract | Reentrancy Type | ECFCHECKER | SEREUM | ÆGIS |
|---|---|---|---|---|
| *VulnBankNoLock* | Same-Function | TP | TP | TP |
| | Cross-Function | FN | TP | TP |
| *VulnBankBuggyLock* | Same-Function | TN | FP | TN |
| | Cross-Function | FN | TP | TP |
| *VulnBankSecureLock* | Same-Function | TN | FP | TN |
| | Cross-Function | TN | FP | TN |

the same pattern as a reentrancy attack and SEREUM being unable to differentiate between these two. We found that ÆGIS correctly classifies every contract as either vulnerable or not vulnerable in all the test cases.

**Unconditional Reentrancy.** Calls that send ether are usually protected by a check in the form of an `if`, `require`, or `assert`. Reentrancy attacks typically try to bypass these checks. However, it is possible to write a contract, which does not perform any check before sending ether. Rodler et al. present an example of such a vulnerability and name it *unconditional reentrancy* (see Figure 9.9). Moreover, they also find an example of such a contract deployed on the Ethereum blockchain[7]. When SEREUM was published, it was not able to detect this type of reentrancy since the authors assumed that every call that may lead to a reentrancy is guarded by a condition. However, the authors claim to have fixed this issue by extending SEREUM to tracking data flows from storage to the parameters of calls. We cannot verify this since the source code of SEREUM is not publicly available. We run ÆGIS on both examples, the manually crafted example by Rodler et al. and the contract deployed on the Ethereum blockchain. ÆGIS correctly identifies the unconditional reentrancy contained in both examples without modifying the existing patterns. This is as expected, since in contrast to SEREUM's initial way to detect reentrancy, ÆGIS's reentrancy patterns do not rely on the detection of conditions (i.e., `JUMPI`) to detect reentrancy.

### 9.4.2 Large-Scale Blockchain Analysis

In this experiment we analyze the first 4.5 million blocks of the Ethereum blockchain and compare our findings to those of Rodler et al. We started by scanning the Ethereum blockchain for smart contracts that have been deployed until block 4,500,000. We found

[7] `https://etherscan.io/address/0xb7c5c5aa4d42967efe906e1b66cb8df9cebf04f7`

```
1  contract VulnBank {
2    mapping (address => uint) public userBalances;
3
4    function deposit() public payable {
5      userBalances[msg.sender] += msg.value;
6    }
7
8    function withdrawAll() public {
9      uint amountToWithdraw = userBalances[msg.sender];
10     msg.sender.call.value(amountToWithdraw)("");
11     userBalances[msg.sender] = 0;
12   }
13 }
```

**Figure 9.9:** Example of a contract that is vulnerable to unconditional reentrancy [166].

675,444 successfully deployed contracts. The deployment timestamps of the found contracts range from August 7, 2015 to November 6, 2017. Next, we replayed the execution history of these 675,444 contracts. As part of the scanning we found that only 12 contracts in our dataset have more than 10,000 transactions. Therefore, to reduce the execution time, we decided to limit our analysis to the first 10,000 transactions of each contract. In addition, similar to Rodler et al., we tried our best to skip those transactions which were involved in denial-of-service attacks as they would result in high execution times[8].

**Table 9.4:** Number of vulnerable contracts detected by ÆGIS.

| Vulnerability | Contracts | Transactions |
|---|---|---|
| Same-Function Reentrancy | 7 | 822 |
| Cross-Function Reentrancy | 5 | 695 |
| Delegated Reentrancy | 0 | 0 |
| Create-Based Reentrancy | 0 | 0 |
| Parity Wallet Hack 1 | 3 | 80 |
| Parity Wallet Hack 2 | 236 | 236 |
| Total Unique | 248 | 1118 |

We ran ÆGIS on our set of 675,444 contracts using a 6-core Intel Core i7-8700 CPU @ 3.20GHz and 64 GB RAM. Our tool took on average 108 milliseconds to analyze a transaction, with a median of 24 milliseconds per transaction. All in all, we re-executed 4,960,424 transactions with an average of 8 transactions per contract. Table 9.4 summarizes our results. ÆGIS found a total of 1,118 malicious transactions and 248 unique contacts that have been exploited through either a reentrancy or an access control vulnerability. More specifically, ÆGIS found that 7 contracts have become victim to same-function reentrancy,

[8] https://tinyurl.com/rvlvues

5 contracts to cross-function reentrancy, 3 contracts to the first Parity wallet hack and 236 contracts to the second Parity wallet hack. Similar to the results of Rodler et al., we did not find any contracts to have become victim to delegated reentrancy or create-based reentrancy. We validated all our results by manually analyzing the source code (whenever it was publicly available) and/or the execution traces of the flagged contracts. Our validation did not reveal any false positives.

**Table 9.5:** Same-function reentrancy vulnerable contracts detected by ÆGIS. Contracts highlighted in gray have only been detected by ÆGIS and not by SEREUM.

| Contract Address | Block Range |
|---|---|
| 0xd654bdd32fc99471455e86c2e7f7d7b6437e9179 | 1680024 - 1680238 |
| 0xbb9bc244d798123fde783fcc1c72d3bb8c189413 | 1718497 - 2106624 |
| 0xf01fe1a15673a5209c94121c45e2121fe2903416 | 1743596 - 1743673 |
| 0x304a554a310c7e546dfe434669c62820b7d83490 | 1881284 - 1881284 |
| 0x59752433dbe28f5aa59b479958689d353b3dee08 | 3160801 - 3160801 |
| 0xbf78025535c98f4c605fbe9eaf672999abf19dc1 | 3694969 - 3695510 |
| 0x26b8af052895080148dabbc1007b3045f023916e | 4108700 - 4108700 |

Table 9.5 lists all the contract addresses that ÆGIS detected to have become victim of a same-function reentrancy attack. The block range defines the block heights where ÆGIS detected the malicious transactions. The first and second contract addresses contained in Table 9.5 are the same as reported by SEREUM, and belong to the DSEthToken and DAO contract, respectively. The rows highlighted in gray mark 5 contracts that have been flagged by ÆGIS but not by SEREUM. After investigating the transactions of these 5 contracts, we find that the contract addresses `0x26b8af052895080148dabbc1007b3045f023916e` and `0xbf78025535c98f4c605fbe9eaf672999abf19dc1` became victim to same-function reentrancy, but seem to be contracts that have been deployed with the purpose of studying the DAO hack. However, the three other contract addresses seem to be true victims of reentrancy attacks.

## 9.5 Discussion

In this section, we discuss alternatives to determine eligible voters, highlight some of the current limitations as well as future research directions for this work.

### 9.5.1 Determining Eligible Voters

The introduction of new patterns in ÆGIS depends on achieving consensus in a predetermined group of voters. Although it may intuitively make sense to let miners vote, they are

not necessarily a good fit. Their interests may differ from those of smart contract users. For example, depending on a pattern's complexity, it might introduce an overhead in terms of execution time. Miners are then incentivized to prefer simpler patterns that are evaluated quicker, while smart contract users would prefer more secure patterns.

Alternatively, a group of trusted security experts could act as eligible voters[9]. Security experts are (by definition) able to properly evaluate patterns and have the interest in doing so. The voting contract is then controlled by a group of trusted experts who are decided upon off-chain by a group of admins. For transparency, the identity of admins and experts would be exposed to the public by mapping every identity to an Ethereum account. Changes to the list of voters, the deposit, or the commit and reveal windows are then visible to anyone via the blockchain. Through this setup, security experts would be able to organize themselves with the voter list being comprised of a curated group of knowledgeable people. Such groups already exist in reality, for example, the members of the Smart Contract Weakness Classification registry (SWC)[10], and would be a good fit for our system. Moreover, misbehaving or unresponsive experts could be easily removed by the group of admins. Although this approach allows for scalability and control, it has the disadvantage of introducing managing third-parties. That runs counter to the decentralized concept of Ethereum.

Alternatively, there is also an option to select voters, while preserving the decentralized concept of Ethereum. This is to remove the role of admins altogether, and instead follow a self-organizing strategy, similar to Proof-of-Stake. In this case, everyone is allowed to become a voter through the purchase of (not prohibitively priced) voting power. This could be achieved by depositing a fixed amount of ether into the voting smart contract as a form of collateral.

### 9.5.2 Adoption and Participation Incentives

The deployment of ÆGIS would require a modification of the Ethereum consensus protocol, which would require existing Ethereum clients to be updated. This could be easily achieved though a major release by including this one-time modification as part of a scheduled hard-fork. Another issue concerns the incentives to propose and vote on patterns. While prestige or a feeling of contributing to the security of Ethereum may be sufficient for some, more incentives may be needed to ensure that the protective capabilities of ÆGIS are used to the full extent. A monetary incentive could address this. That is, ÆGIS could be extended with automatically paid rewards. In other words, ÆGIS could be extended to enable bug bounties [88]. ÆGIS's smart contract could be modified such that, owners of smart contracts can register their contract address by sending a transaction to ÆGIS's voting smart contract and deposit a bounty in the form of ether. Then, proposers of patterns would be rewarded automatically with the bounty by ÆGIS's voting smart contract, if their proposed

[9] Somewhat similar to how CVEs are handled.

[10] `https://smartcontractsecurity.github.io/SWC-registry/`

pattern is accepted by the group of voters. Moreover, owners could simply replenish the bounty for their contract by making new deposits to ÆGIS's smart contract.

### 9.5.3 Limitations

A current limitation of our tool is that proposed attack patterns are submitted in plain text to the smart contract. Potential attackers can view the patterns and use them to find vulnerable smart contracts. To mitigate this, we propose to make use of encryption such that only the voters would be able to view the patterns. However, this would break the current capability of the smart contract being self-tallying. Designing an encrypted and practical self-tallying solution is left for future work. Finally, we intend to make use of parallel execution inside the extractors and the checking of patterns in order to improve the time required to analyze transactions.

## 9.6 Related Work

As with any program, smart contracts may contain bugs and can be vulnerable to exploitation. As discussed in [60], different types of vulnerabilities exist, often leading to financial losses. The issue is made worse by the fact that smart contracts are immutable. Once deployed, they cannot be altered and vulnerabilities cannot be fixed. In addition to that, automated tools for launching attacks exist [116].

Several defense mechanisms have been proposed to detect security vulnerabilities in smart contracts. This includes tools such as ERAYS [140], designed to provide smart contract auditors with a reverse engineered pseudo code of a contract from its bytecode. The interpretation of the pseudo code however remains a slow and grueling task. More automated tools have also been proposed benefiting from regular expressions [172] and machine learning techniques [134] to detect vulnerabilities.

A wealth of security research has focused on the creation of static analysis tools to automatically detect vulnerabilities in smart contracts. Formal verification has been used together with a formal definition of the EVM [107, 85], or by first converting smart contracts into the formal language F* [35, 104]. Other works focused on analyzing the higher level solidity code [135, 155], which limits the scope to those contracts with available source code. Another approach is to apply static analysis on the smart contract bytecode [136]. A technique commonly used for this purpose is symbolic execution, designed to thoroughly explore the state space of a smart contract utilizing constraint solving. It has been used to detect contracts with vulnerabilities [51, 188], to find misbehaving contracts [122, 115, 158], or detect integer bugs [102, 112]. Fuzzing techniques have also been applied [110, 160]. In [171] the authors propose HARVEY, a greybox fuzzer that selects appropriate inputs and transaction sequences to increase code coverage. Fuzzing techniques however involve a trade-off between the number of discovered paths and the efficiency in input generation.

While all the listed tools help identify vulnerabilities, they cannot protect already deployed smart contracts from being exploited. Therefore, to deal with the issue of vulnerabilities in deployed smart contracts, [69, 165] propose a modification to the Ethereum client, that would allow detection and prevent exploitation of reentrancy vulnerabilities at runtime. However, these approaches only deal with reentrancy and require all the clients in the network to be modified. This is an issue for the following reasons. On one hand, every update of the vulnerability detection software requires an update of the different Ethereum client implementations. This is true for both bug fixes and functionality upgrades, for example the detection of new vulnerabilities. On the other hand, every modification of the clients needs to be adopted by all the nodes participating in the Ethereum blockchain. This requires time and breaks compatibility between updated and non-updated clients. In this work, we propose a generic solution that protects contracts and users from existing and future vulnerabilities, without requiring client modifications and forks every time a new vulnerable smart contract is found.

Wang et al. [169] propose an approach to detect vulnerabilities at runtime based on two invariants that follow the intuition that most vulnerabilities are due to a mismatch between the transferred amount and the amount reflected by the contract's internal bookkeeping logic. However, this approach has three main drawbacks. First, it requires the automated and correct identification of bookkeeping variables, which besides being a non-trivial task also does not hold for every contract, since there can be contracts that do not use internal bookkeeping logic but are nevertheless vulnerable. Second, their approach does not model environmental information such as timestamps or block numbers, which does not allow them to detect vulnerabilities such as timestamp dependence or transaction order dependency, whereas our approach models environmental information and allows for the detection of these vulnerabilities. Finally, Wang et al.'s approach can only detect violations of safety properties and not violations of liveness properties such as the Parity Wallet Hack 2. In this work, we demonstrate that our approach is capable of detecting both Parity wallet hacks and therefore violations to safety as well as liveness properties.

As blockchains provide means for transparency and decentralization, multiple blockchain-based solutions have been proposed to perform electronic voting [53, 61, 108]. With the recent developments in quantum computers, recent work has also started to focus on the development of quantum-resistant blockchain-based voting schemes [168]. These solutions can all be categorized into two categories: cryptocurrency-based and smart-contract-based.

Cryptocurrency-based solutions focus on using payments as a proxy for votes in an election. When a voter wishes to cast a vote, he or she makes a payment to the address of the candidate. Lee et al. [50] proposed such a system in the Bitcoin network. However, their system requires a trusted third party to perform the ballot counting. Zao et al. [33] were the first to propose a voting scheme using the public Bitcoin network while preserving the privacy of the votes. Another well-known cryptocurrency-based solution is CarbonVote [39]. It was introduced in the aftermath of the DAO hack to allow the Ethereum Foundation to

determine if the Ethereum community wanted a hard fork or not. The tallying was performed by counting the amount of ether that each address received. Needless to say, such a system gives a tremendous amount of voting power to users with a large amount of funds.

Smart-contract-based voting relies on a decentralized application to assist the voting process – there is no central entity. McCorry et al. [77] propose a practical implementation of the Open Vote Network [18] in the form of a smart contract deployed on the Ethereum blockchain for boardroom voting. Their implementation is self-tallying and provides, in addition to vote privacy, also transparency. Voting proceeds in several rounds, where the voters first broadcast their voting key, followed by a proof that their vote is binary (a “yes” or “no” vote). A final tally round allows anyone to calculate the total sum of votes, without revealing individual ballots. The voting mechanism described in this chapter is inspired by McCorry et al.’s proposed solution and implementation. The limitations of their proposed solution, namely having a binary voting system and limiting the number of voters to less than 50 participants, are acceptable for our purposes.

## 9.7 Conclusion

Despite various researchers having proposed a significant number of tools to detect vulnerabilities in smart contracts, only a small number have proposed a solution to protect deployed smart contracts against attacks. These solutions depend on the modification of the Ethereum clients in order to detect and revert transactions that try to exploit vulnerable smart contracts. However, these solutions require all the Ethereum clients to be modified every time a new type of vulnerability is discovered. In this chapter, we introduced ÆGIS, a system that detects and reverts attacks via attack patterns. These patterns describe malicious control and data flows through the use of a novel domain-specific language. In addition, we presented a novel mechanism for security updates that allows these attack patterns to be updated quickly and transparently via the blockchain, by using a smart contract as means of storing them. Finally, we compared ÆGIS to two current state-of-the-art online reentrancy detection tools. Our results show that ÆGIS not only detects more attacks, but also has no false positives as compared to current state-of-the-art.

# 10 | Conclusions

In this dissertation, we studied the topic of analyzing and improving the security of smart contracts. Today, five years after the DAO hack, attacks on smart contracts still remain a pertinent issue. Years of research on smart contract security attribute to the fact that attacks are now more sophisticated than ever. In this dissertation, we studied the security of smart contracts by presenting methods to automatically detect vulnerabilities and attacks in smart contracts, and tried to conceive defense mechanisms that focus on preventing exploits that stem from well-studied smart contract vulnerabilities. To that end, we studied the domain of smart contract security from three different angles: (1) automated vulnerability detection, (2) investigation of attacks, and (3) pre- and post-deployment defenses.

First, we explored the use of static and dynamic analysis methods to perform automated vulnerability detection on smart contract bytecode. In Chapter 3, we investigated whether integer overflows are an apparent issue in smart contracts. In that light, we presented OSIRIS – a symbolic execution tool for detecting integer bugs such as arithmetic bugs, truncation bugs, and signedness bugs, in Ethereum smart contracts. To reduce false positives, our symbolic execution tool leverages taint analysis to only report integer bugs that can be triggered by attackers and which flow into sensitive program locations such as a call to another contract or a write to storage. Our comparison showed that current state-of-the-art is in fact not sound as sometimes claimed and that our methodology is able to report less false positives. Moreover, our results stipulate that there exist a significant number of deployed smart contracts that are vulnerable to integer bugs. Therefore, we identified causes for integer bugs and proposed modifications to the EVM and the Solidity compiler. However, the downside of symbolic execution is that it is notorious for producing false positives and that it suffers from the path explosion problem as programs become more complex. Consequently, in Chapter 4, we investigated the use of hybrid fuzzing as an alternative to symbolic execution for smart contracts. We presented CONFUZZIUS, the first hybrid fuzzer for smart contracts which tackles the three main challenges of smart contract testing: *input generation*, *stateful exploration*, and *environmental dependencies*. We solved the first challenge by combining evolutionary fuzzing with constraint solving to generate inputs that allow the fuzzer to get past complex path conditions. Finally, we solved the last challenge by leveraging data dependency analysis across state variables to generate meaningful transaction

sequences instead of random ones. The last challenge, we solved by modeling block related information (e.g., block number) and contract related information (e.g., call return values) as fuzzable inputs. We evaluated the performance of our hybrid fuzzer by comparing it to other state-of-the-art fuzzers and symbolic execution tools for smart contracts and demonstrated that our hybrid fuzzer is able to detect more bugs and achieve more code coverage than existing state-of-the-art tools.

Next, we focused on the investigation of attacks against smart contracts. In Chapter 5, we presented the design and implementation of HORUS – an extensible framework for detecting, analyzing, and tracing smart contract attacks. Our framework identified thousands of attacks on real-world deployed smart contracts by analyzing transactions of a 4.5 year period. In particular, we found that the number of attacks seem to have decreased for attacks such as integer overflows, whereas the number of attacks related to unhandled exceptions and reentrancy seem to remain a pertinent issue despite a large number of freely available smart contract security tools. Moreover, by using two recent examples of attacks on decentralized exchanges, we demonstrated how our framework can be leveraged by researchers and companies to perform in-depth post-mortem analyses of smart contract incidents. Further, smart contracts may not only be victims of attacks, they may also be used by fraudsters to mount attacks. In Chapter 6, we investigated an emerging new type of fraud in Ethereum called *honeypots*. We created a taxonomy of honeypot techniques and build a tool called HONEYBADGER, that leverages symbolic execution together with well-defined heuristics to automatically detect honeypots. We showed that our methodology can effectively detect honeypots with a very low false positive rate. In a large-scale experiment, we were able to identify several honeypot contracts in the wild. Our analysis on a subset of identified honeypots, revealed that already several users fell for these honeypots and that attackers already made a significant amount of profit by deploying these contracts. However, honeypots are not the only type of fraud where smart contracts can be used to make profit on the back of innocent users. The rise of decentralized finance stimulated a number of attackers to perform so-called *frontrunning* attacks against smart contracts. We investigated the prevalence of these attacks, by presenting a methodology to efficiently measure the three different types of frontrunning: *displacement*, *insertion*, and *suppression*, while relying only on historical information from the blockchain. Our analysis identified thousands of attacks in the wild, thereby providing evidence that frontrunning is both, lucrative and a prevalent issue, that has serious implications for modern blockchains.

Finally, we studied pre- and post-deployment defenses for smart contracts. To protect smart contracts against attacks before deployment, we proposed in Chapter 8 ELYSIUM – a tool to automatically patch vulnerable smart contracts using context-related information that is inferred at the bytecode level. Our tool is currently able to patch 7 types of vulnerabilities and can easily be extended by adding further vulnerability detectors or by writing new patch templates using our custom domain-specific language. We evaluated our tool and demon-

strated that our approach of combining template-based with semantic-based patching is able of effectively and correctly patch more contracts than existing tools, with a minimal increase in transaction and deployment costs. However, being able to automatically patch vulnerable smart contracts prior deployment does not protect smart contracts that have already been deployed. To that end, in Chapter 9 we introduced ÆGIS – a system that detects attacks via attack patterns and neutralizes them by reverting their effects. These patterns can be written by security experts using a domain-specific language that is dedicated to the detection of malicious control- and data-flows inside the Ethereum virtual machine. Moreover, the system proposes a novel mechanism for security updates by storing attack patterns inside a smart contract. This mechanism allows attack patterns to be updated quickly and transparently via the blockchain. We compared our system to two current state-of-the-art online reentrancy detection tools and show that our system detects more attacks in comparison to current state-of-the-art tools while achieving zero false positives.

In summary, this dissertation provides a variety of tools and frameworks that academia as well as industry may use not only to detect vulnerabilities in smart contracts, but also to mount defenses against smart contract attacks. For example, developers can use OSIRIS and CONFUZZIUS to detect vulnerabilities during development and patch them before deployment using ELYSIUM. After deployment, developers can use ÆGIS to protect their smart contracts against attacks and leverage HORUS to monitor and trace the flow of stolen assets in case their smart contracts become compromised. However, the presented tools and frameworks also have their own limitations and security is in general a cat-and-mouse game, where attackers constantly seek to find new sophisticated ways to dismantle the security of software systems and where security experts have to come up with new ways to defend against these sophisticated attacks. Nonetheless, the results presented in this dissertation shed some light on the security of smart contracts and provide methods and tools that can be used as building blocks for future research directions.

## Future Directions

Despite exploring the security of smart contracts from three different angles, this dissertation only scratches the surface of smart contract security. Research on the topic of analyzing and improving the security of smart contracts remains relevant as long as smart contracts continue to suffer from attacks. In the following, we present possible directions in which the work presented in this dissertation could be extended.

**Composability of Smart Contracts.** One possible future research direction is to conceive new techniques to detect vulnerabilities that emerge from the composability of smart contract protocols. With the rise of decentralized finance, smart contracts became complex software systems that heavily interact with a variety of smart contracts. Prior to the decentralized finance revolution, smart contracts were merely interacting with one or two contracts.

Nowadays, people can borrow, invest, and trade across a multitude of smart contracts within a single transaction. Due to this composability, decentralized finance is often described as "money lego". Unfortunately, this composability also introduces new security challenges and requires a definition of a new threat model. Smart contracts that were previously considered secure are now vulnerable to attacks when combined with other contracts in unexpected ways. Prominent examples include the previously analyzed Uniswap and Lendf.me hacks in Chapter 5, but also more recent examples such as the Cream Finance hacks in 2021 that resulted together in assets worth almost 200M USD being stolen [205]. There is a pressing need for tools that can detect flaws across smart contract protocols. One possible solution would be to perform formal verification by leveraging symbolic execution techniques as used in Chapters 3 and 6, and extend them to perform inter-contract analysis on smart contracts. However, this requires the correct modeling of call chains and new strategies to efficiently deal with the exhaustive number of possible combinations of function calls. Another solution would be to add inter-contract analysis to our hybrid fuzzer that we presented in Chapter 4. Hence, instead of testing only one smart contract at a time and treating external calls as symbolic values, we could perform concrete execution on external calls and test the effects of calling other smart contracts. However, also in this case inter-contract analysis will lead to a large number of function combinations to be analyzed. Thus, optimizations will be required to make inter-contractual analysis feasible in practice. Fortunately, our fuzzer already provides a way to prioritize the combination of function calls by solely combining functions that share data dependencies across state variables and would only need to be adapted to identify data dependencies across smart contracts.

**Robustness of Ethereum WebAssembly.** Ethereum is planning to replace its current execution model with an Ethereum flavored version of WebAssembly (Wasm) called EWasm. EWasm is a restricted subset of Wasm, that has been customized for the execution of Ethereum smart contracts. EWasm will enable the execution of smart contracts to be near-native speed, but it will also introduce new challenges. For example, Ethereum currently enforces the termination of smart contracts via a gas model. The model associates a certain gas cost to the execution of each instruction. This guarantees that the execution will always stop once no more gas is left to pay for the execution. The challenge will be to propose a new fee schedule for EWasm that does not result in resource exhaustion attacks due to certain instructions being underpriced. Ethereum has already faced several denial-of-service attacks in the past due to underestimated gas costs. The existing pricing model uses a static attribution of gas costs, meaning that the gas costs have been manually assigned to the instructions. Hence, one possible future direction, would be to analyze the proposed fee schedule for EWasm and to verify that attackers will not be able to mount denial-of-service attacks against EWasm-enabled smart contracts, and to propose a better pricing model that takes into account the various client implementations and dynamically assigns a gas cost to each instruction which reflects its resource usage (e.g., IO, CPU, memory, etc.).

**Fair Ordering of Transactions.** In Chapter 7, we analyzed the prevalence of transaction ordering attacks on the Ethereum blockchain. These attacks pose a serious threat to the adoption of blockchain technology as a whole. One possible future direction, would be to come up with new strategies to either establish a fair ordering of transactions or to provide transaction privacy. A fair ordering would ideally mean that a user who sent his transaction first, will be also executed first. One solution would be to have a globally synchronized clock across all the clients. Unfortunately this is a property that is extremely hard to achieve in practice, especially in decentralized peer-to-peer networks of thousands of nodes. An alternative solution to fair ordering, could be the encryption of transaction content, including the encryption of smart contracts storage, and to force miners to execute encrypted transactions inside a trusted execution environment (TEE). Miners should not be able to inspect the outcome of an execution or care about the contents of a transaction. Moreover, other nodes should not be able to inspect the contents of pending transactions that do not belong to them. The encryption of transactions and smart contract storage would guarantee confidentially of data in transit and at rest, while TEEs would guarantee confidentially of data at execution. Thus, attackers would not know which transactions to manipulate, since the contents and outcome of transactions would remain secret, even to miners. The issue here is the distribution of encryption keys across a public and decentralized peer-to-peer network where nodes can simply join and leave at any time.

**Control- and Data-Flow Integrity for Smart Contracts.** In Chapter 8, we proposed a defense for smart contracts by automatically patching vulnerable bytecode prior to deployment using context-sensitive patching. However, another possible future research direction would be to investigate the applicability of control- and data-flow integrity solutions as a defense mechanism for smart contracts. The solutions proposed in Chapters 8 and 9 follow a blacklisting approach, meaning that they try to define what malicious control- and data-flows are in order to block them. The issue with such an approach is the lack of generalizability. New vulnerabilities will require the definition of new malicious control- and data-flows. A way to alleviate this cat-and-mouse game, is to follow a whitelisting approach, where the idea is to analyze the interactions between users and contracts in order to learn what legitimate control- and data-flows are and block executions that divert from these “regular” flows. Integrity checks would help in preventing unforeseen combinations of smart contracts calls that might result in reentrancy attacks or calls to privileged functions. The challenge here would be the inference of “regular” flows and to design an approach that does not impede legitimate execution while being efficient enough in terms of execution time, bytecode size, and gas usage to monitor, analyze, and verify the legality of execution paths at runtime.

# Publications


[102] Christof Ferreira Torres, Julian Schütte, and Radu State. “Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts”. In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 664–676.

[157] Christof Ferreira Torres, Mathis Baden, Robert Norvill, and Hugo Jonker. “ÆGIS: Smart Shielding of Smart Contracts (Poster)”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 2019, pp. 2589–2591.

[158] Christof Ferreira Torres, Mathis Steichen, and Radu State. “The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts”. In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association, 2019, pp. 1591–1607.

[180] Christof Ferreira Torres, Mathis Baden, Robert Norvill, Beltran Borja Fiz Pontiveros, Hugo Jonker, and Sjouke Mauw. “ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks”. In: *ASIA CCS ’20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*. ACM, 2020, pp. 584–597.

[209] Christof Ferreira Torres, Ramiro Camino, and Radu State. “Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. USENIX Association, 2021, pp. 1343–1359.

[210] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. “ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts”. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 103–119.

[211] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. “The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts”. In: *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part I*. Vol. 12674. Lecture Notes in Computer Science. Springer, 2021, pp. 33–52.

[222] Christof Ferreira Torres, Hugo Jonker, and Radu State. “Elysium: Automagically Healing Vulnerable Smart Contracts Using Context-Aware Patching”. In: *arXiv preprint arXiv:2108.10071* (2021).

[226] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc Viet Le, and Arthur Gervais. “High-Frequency Trading on Decentralized On-Chain Exchanges”. In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 428–445.

# Bibliography


[1] Burton H Bloom. “Space/Time Trade-Offs in Hash Coding with Allowable Errors”. In: *Communications of the ACM* (1970), pp. 422–426.

[2] David Chaum, Amos Fiat, and Moni Naor. “Untraceable electronic cash”. In: *Conference on the Theory and Application of Cryptography*. Springer. 1988, pp. 319–327.

[3] Gunar E Liepins and Michael D Vose. “Representational issues in genetic optimization”. In: *Journal of Experimental & Theoretical Artificial Intelligence* (1990), pp. 101–115.

[4] Barton P Miller, Louis Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[5] Bill Cheswick. “An Evening with Berferd in which a cracker is Lured, Endured, and Studied”. In: *In Proc. Winter USENIX Conference*. 1992, pp. 163–174.

[7] Wei Dai. *B-money*. 1998. URL: http://www.weidai.com/bmoney.txt.

[8] Edsger W Dijkstra. “Solution of a problem in concurrent programming control”. In: *Pioneers and Their Contributions to Software Engineering*. Springer, 2001, pp. 289–294.

[9] Li Yujian and Liu Bo. “A normalized Levenshtein distance metric”. In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1091–1095.

[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[11] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[12] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. “Automated Whitebox Fuzz Testing.” In: *NDSS*. Vol. 8. 2008.

[13] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: (2008). URL: https://www.metzdowd.com/pipermail/cryptography/2008-October/014810.html.

[14] Ping Chen, Hao Han, Yi Wang, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. “IntFinder: Automatically detecting integer bugs in x86 binary program”. In: *International Conference on Information and Communications Security*. Springer. 2009, pp. 336–345.

[15] David Molnar, Xue Cong Li, and David Wagner. “Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs.” In: *USENIX Security Symposium*. Vol. 9. 2009, pp. 67–82.

[16] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. “IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution”. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. 2009.

[17] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. “Automatically finding patches using genetic programming”. In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 364–374.

[18] Feng Hao, Peter YA Ryan, and Piotr Zieliński. “Anonymous voting by two-round public discussion”. In: *IET Information Security* 4.2 (2010), pp. 62–67.

[19] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[20] R Sivaraj and T Ravichandran. “A review of selection methods in genetic algorithm”. In: *International journal of engineering science and technology* 3.5 (2011), pp. 3792–3797.

[21] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. “Improving Integer Security for Systems with KINT.” In: *OSDI*. Vol. 12. 2012, pp. 163–177.

[22] Tyler Moore and Nicolas Christin. “Beware the middleman: Empirical analysis of Bitcoin-exchange risk”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2013, pp. 25–33.

[23] Malte Moser, Rainer Bohme, and Dominic Breuker. “An inquiry into money laundering tools in the Bitcoin ecosystem”. In: *eCrime Researchers Summit (eCRS), 2013*. IEEE. 2013, pp. 1–14.

[24] Vitalik Buterin. *Ethereum Whitepaper*. Jan. 2013. URL: `https://ethereum.org/en/whitepaper/`.

[25] Ittay Eyal and Emin Gün Sirer. “Majority is not enough: Bitcoin mining is vulnerable”. In: *International conference on financial cryptography and data security*. Springer. 2014, pp. 436–454.

[26] Marios Pomonis, Theofilos Petsios, Kangkook Jee, Michalis Polychronakis, and Angelos D Keromytis. “IntFlow: improving the accuracy of arithmetic error detection using information flow tracking”. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM. 2014, pp. 416–425.

[27] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. “The strength of random search on automated program repair”. In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 254–265.

[28] Aaron Reeves, Martin McKee, and David Stuckler. “Economic suicides in the great recession in Europe and North America”. In: *The British Journal of Psychiatry* 205.3 (2014), pp. 246–247.

[29] Rosalind Wiggins, Thomas Piontek, and Andrew Metrick. “The Lehman brothers bankruptcy a: overview”. In: *Yale program on financial stability case study* (2014).

[30] Gavin Wood. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum Project Yellow Paper* 151 (2014), pp. 1–32.

[31] Marie Vasek and Tyler Moore. “There’s no free lunch, even using Bitcoin: Tracking the popularity and profits of virtual currency scams”. In: *International conference on financial cryptography and data security*. Springer. 2015, pp. 44–61.

[32] Fabian Vogelsteller and Vitalik Buterin. *ERC-20 Token Standard*. Feb. 2015. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[33] Zhichao Zhao and T.-H. Hubert Chan. “How to Vote Privately Using Bitcoin”. In: *Proc. 17th International Conference on Information and Communications Security (ICICS’15)*. Ed. by Sihan Qing, Eiji Okamoto, Kwangjo Kim, and Dongmei Liu. Lecture Notes in Computer Science. Springer, 2015, pp. 82–96.

[35] Karthikeyan Bhargavan et al. “Formal Verification of Smart Contracts: Short Paper”. In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. PLAS ’16. ACM, 2016, pp. 91–96. ISBN: 978-1-4503-4574-3.

[36] Joseph Bonneau. “Why buy when you can rent?” In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 19–26.

[37] Vitalik Buterin. *CRITICAL UPDATE Re: DAO Vulnerability*. 2016. URL: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/.

[38] ConsenSys. *Ethereum Smart Contract Best Practices – Known Attacks*. June 2016. URL: https://consensys.github.io/smart-contract-best-practices/known_attacks.

[39] Ashu Daniel Lv. *CarbonVote*. 2016. URL: https://http://carbonvote.com/.

[40] Ethererik. *GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas*. Apr. 2016. URL: https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/.

[41] Etherscan. *The DAO*. 2016. URL: https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.

[42] Etherscan. *The Dark DAO*. 2016. URL: https://etherscan.io/address/0x304a554a310c7e546dfe434669c62820b7d83490#code.

[43] Etherscan. *The Run*. Apr. 2016. URL: https://etherscan.io/address/0xcac337492149bdb66b088bf5914bedfbf78ccc18%5C#code.

[44] Yoichi Hirai. *Exception on overflow - Issue #796 - ethereum/solidity*. June 2016. URL: https://github.com/ethereum/solidity/issues/796#issuecomment-253578925.

[45] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. "Soufflé: On synthesis of program analyzers". In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 422–430.

[46] King of the Ether Throne. *KotET – Post-Mortem Investigation*. Feb. 2016. URL: https://www.kingoftheether.com/postmortem.html.

[47] Klint Finley. *A $50 Million Hack Just Showed That the DAO Was All Too Human*. June 2016. URL: https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/.

[48] Felix Lange. *Security Alert – DoS Vulnerability in the Soft Fork*. 2016. URL: https://blog.ethereum.org/2016/06/28/security-alert-dos-vulnerability-in-the-soft-fork/.

[49] ledgerwatch. *I think TheDAO is getting drained right now*. 2016. URL: https://www.reddit.com/r/ethereum/comments/4oi2ta/i_think_thedao_is_getting_drained_right_now/.

[50] Kibin Lee, Joshua I James, Tekachew G Ejeta, and Hyoung J Kim. "Electronic voting service using block-chain". In: *Journal of Digital Forensics, Security and Law* 11.2 (2016), p. 8.

[52] Michal Zalewski. *American Fuzzy Lop (AFL)*. Dec. 2016. URL: http://lcamtuf.coredump.cx/afl/.

[53] Ryan Osgood. "The future of democracy: Blockchain voting". In: *COMP116: Information Security* (2016), pp. 1–21.

[54] Ramsroyal. *Why is my node synchronization stuck/extremely slow at block 2,30 6,843?* Nov. 2016. URL: https://ethereum.stackexchange.com/questions/9883/why-is-my-node-synchronization-stuck-extremely-slow-at-block-2-306-843.

[55] Christian Reitwiessner. *Smart Contract Security*. 2016. URL: https://blog.ethereum.org/2016/06/10/smart-contract-security/.

[56] Kosta Serebryany. “Continuous fuzzing with libfuzzer and addresssanitizer”. In: *2016 IEEE Cybersecurity Development (SecDev)*. IEEE. 2016, pp. 157–157.

[57] David Siegel. *Understanding The DAO Attack*. June 2016. URL: https://www.coindesk.com/understanding-dao-hack-journalists/.

[58] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[59] Stephan Tual. *No DAO funds at risk following the Ethereum smart contract ’recursive call’ bug discovery*. 2016. URL: https://medium.com/ursium-blog/no-dao-funds-at-risk-following-the-ethereum-smart-contract-recursive-call-bug-discovery-29f482d348b.

[60] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. “A Survey of Attacks on Ethereum Smart Contracts (SoK)”. In: *Proc. 6th International Conference on Principles of Security and Trust - Volume 10204*. Vol. 10204. Lecture Notes in Computer Science. Springer-Verlag, 2017, pp. 164–186. ISBN: 978-3-662-54454-9.

[61] Ahmed Ben Ayed. “A conceptual secure blockchain-based electronic voting system”. In: *International Journal of Network Security & Its Applications* 9.3 (2017), pp. 01–09.

[62] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. “Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact”. In: *arXiv preprint arXiv:1703.03779* (2017).

[63] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. “Directed greybox fuzzing”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2329–2344.

[64] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-based greybox fuzzing as markov chain”. In: *IEEE Transactions on Software Engineering* 45.5 (2017), pp. 489–506.

[65] Michael J. Coblenz. “Obsidian: a safer blockchain programming language”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. IEEE Computer Society, 2017, pp. 97–99.

[66] devops199. *anyone can kill your contract #6995*. 2017. URL: https://github.com/paritytech/parity-ethereum/issues/6995.

[67] Etherscan. *DSEthToken*. 2017. URL: https://etherscan.io/address/0xd654bdd32fc99471455e86c2e7f7d7b6437e9179#code.

[68] Etherscan. *SmartBillions*. Sept. 2017. URL: https://etherscan.io/address/0x5ace17f87c7391e5792a7683069a8025b83%5C%5Cbbd85%5C#code.

[69] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. "Online detection of effectively callback free objects with applications to smart contracts". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), p. 48.

[70] Yoichi Hirai. "Defining the ethereum virtual machine for interactive theorem provers". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 520–535.

[71] Yoichi Hirai. *Ethereum Virtual Machine for Coq (v0.0.2)*. June 2017. URL: https://medium.com/@pirapira/ethereum-virtual-machine-for-coq-v0-0-2-d2568e068b18.

[72] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. "Chizpurfle: A gray-box android fuzzer for vendor service customizations". In: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2017, pp. 1–11.

[73] Ivan Bogatyy. *Implementing Ethereum trading front-runs on the Bancor exchange in Python*. Oct. 2017. URL: https://medium.com/hackernoon/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798.

[74] Jacques Dafflon, Jordi Baylina, Thomas Shababi. *EIP 777: ERC777 Token Standard*. Nov. 2017. URL: https://eips.ethereum.org/EIPS/eip-777.

[75] LLL. *Ethereum Low-level Lisp-like Language*. July 2017. URL: https://lll-docs.readthedocs.io/en/latest/.

[76] Loi Luu. *Oyente - An Analysis Tool for Smart Contracts v0.2.7 (Commonwealth)*. Feb. 2017. URL: https://github.com/melonproject/oyente.

[77] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. "A Smart Contract for Boardroom Voting with Maximum Voter Privacy". In: *Proc. 21st International Conference on Financial Cryptography and Data Security (FC'17)*. Vol. 10322. Lecture Notes in Computer Science. Springer, 2017, pp. 357–375.

[78] Nooku. *Worry-some bug/exploit with ERC20 token transactions from exchanges*. Apr. 2017. URL: https://www.reddit.com/r/ethereum/comments/63s917/worrysome_bug_exploit_with_erc20_token/dfwmhc3/.

[79] Paweł Bylica. *How to Find $10M Just by Reading the Blockchain*. Apr. 2017. URL: https://medium.com/golem-project/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95.

[80] Sergey Petrov. *Another Parity Wallet hack explained*. Nov. 2017. URL: https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c.

[81] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing." In: *NDSS*. Vol. 17. 2017, pp. 1–14.

[82] Emin Gün Sirer and Phil Daian. *Bancor Is Flawed*. June 2017. URL: https://hackingdistributed.com/2017/06/19/bancor-is-flawed.

[83] Supr3m. *SmartBillions lottery contract just got hacked!* Oct. 2017. URL: https://www.reddit.com/r/ethereum/comments/74d3dc/smartbillions_lottery_contract_just_got_hacked/.

[84] Wolfie Zhao. *$30 Million: Ether Reported Stolen Due to Parity Wallet Breach*. July 2017. URL: https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach.

[85] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. "Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL". In: *CPP. ACM. To appear* (2018).

[86] Massimo Bartoletti, Barbara Pes, and Sergio Serusi. "Data mining for detecting Bitcoin Ponzi schemes". In: *arXiv preprint arXiv:1803.00646* (2018).

[87] Trail of Bits. *Manticore - Symbolic execution tool*. June 2018. URL: https://github.com/trailofbits/manticore.

[88] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. "Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1335–1352.

[89] Lorenz Breidenbach, Tyler Kell, Stephane Gosselin, and Shayan Eskandari. *LibSubmarine – Defeat Front-Running on Ethereum*. 2018. URL: https://libsubmarine.org/.

[90] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. "Vandal: A scalable security analysis framework for smart contracts". In: *arXiv preprint arXiv:1809.03981* (2018).

[91] Peng Chen and Hao Chen. "Angora: Efficient fuzzing by principled search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 711–725.

[92] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. “Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology”. In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2018, pp. 1409–1418.

[93] Cornell Blockchain. *Bamboo: a language for morphing smart contracts*. May 2018. URL: https://github.com/CornellBlockchain/bamboo.

[94] Etherscan. *BeautyChainToken*. June 2018. URL: https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d#code.

[95] Etherscan. *Ethereum Contracts with Verified Source Codes*. June 2018. URL: https://etherscan.io/contractsVerified.

[96] Etherscan. *Etherscan Token Tracker Page*. June 2018. URL: https://etherscan.io/tokens.

[97] Etherscan. *HexagonToken*. June 2018. URL: https://etherscan.io/address/0xb5335e24d0ab29c190ab8c2b459238da1153ceba#code.

[98] Etherscan. *HODLWallet*. 2018. URL: https://etherscan.io/address/0x4a8d3a662e0fd6a8bd39ed0f91e4c1b729c81a38#code.

[99] Etherscan. *SmartMeshICO*. June 2018. URL: https://etherscan.io/address/0x55f93985431fc9304077687a35a1ba103dc1e081#code.

[100] Etherscan. *Social Chain*. June 2018. URL: https://etherscan.io/address/0xb75a5e36cc668bc8fe468e8f272cd4a0fd0fd773#code.

[101] Etherscan. *UselessEthereumToken*. June 2018. URL: https://etherscan.io/address/0x27f706edde3ad952ef647dd67e24e38cd0803dd6#code.

[103] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. “Madmax: surviving out-of-gas conditions in Ethereum smart contracts”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 116.

[104] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. “A Semantic Framework for the Security Analysis of Ethereum Smart Contracts”. In: *Proc. 7th International Conference on Principles of Security and Trust (POST’18)*. Vol. 10804. Lecture Notes in Computer Science. Springer, 2018, pp. 243–269.

[105] NCC Group. *DASP - TOP 10*. 2018. URL: https://dasp.co/.

[106] HeartBank. *Smart Contract Design Patterns*. Oct. 2018. URL: https://medium.com/heartbankstudio/smart-contract-design-patterns-8b7ca8b80dfb.

[107] Everett Hildenbrandt et al. “Kevm: A complete formal semantics of the ethereum virtual machine”. In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.

[108] Friorik P. Hjalmarsson, Gunnlaugur K. Hreioarsson, Mohammad Hamdaqa, and Gísli Hjálmtýsson. “Blockchain-Based E-Voting System”. In: *11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018*. IEEE Computer Society, 2018, pp. 983–986.

[109] PeckShield Inc. *PeckShield Inc. - Advisories*. June 2018. URL: https://peckshield.com/advisories.html.

[110] Bo Jiang, Ye Liu, and WK Chan. “Contractfuzzer: Fuzzing smart contracts for vulnerability detection”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM. 2018, pp. 259–269.

[111] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. *Zeus Evaluation*. 2018. URL: https://docs.google.com/spreadsheets/d/12_g-pKsCtp3lUmT2AXngsqkBGSEoE6xNH51e-of_Za8/preview?usp=embed_googleplus#gid=1568997501.

[112] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. “Zeus: Analyzing safety of smart contracts”. In: NDSS. 2018.

[113] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. “Evaluating fuzz testing”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2123–2138.

[114] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. *CALYPSO: Private Data Management for Decentralized Ledgers*. Tech. rep. Cryptology ePrint Archive, Report 2018/209., 2018.

[115] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. “Exploiting The Laws of Order in Smart Contracts”. In: *arXiv preprint arXiv:1810.11605* (2018).

[116] Johannes Krupp and Christian Rossow. “teether: Gnawing at ethereum to automatically exploit smart contracts”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1317–1333.

[117] Anastasia Mavridou and Aron Laszka. “Tool demonstration: FSolidM for designing secure Ethereum smart contracts”. In: *International Conference on Principles of Security and Trust*. Springer. 2018, pp. 270–277.

[118] misterch0c. *Solidity vulnerable honeypots*. Apr. 2018. URL: https://github.com/misterch0c/Solidlity-Vulnerable/tree/master/honeypots.

[119] Bernhard Mueller. *Detecting Integer Overflows in Ethereum Smart Contracts*. June 2018. URL: https://bit.ly/2JIp9ea.

[120] Bernhard Mueller. “Smashing ethereum smart contracts for fun and real profit”. In: *9th Annual HITB Security Conference (HITBSecConf)*. Vol. 54. 2018.

[121] Paul Muntean, Jens Grosklags, and Claudia Eckert. “Practical Integer Overflow Prevention”. In: *In IEEE TSE journal* (2018). URL: https://arxiv.org/abs/1710.03720.

[122] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. “Finding The Greedy, Prodigal, and Suicidal Contracts at Scale”. In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 653–663.

[123] Onur Solmaz. *The Anatomy of a Block Stuffing Attack*. Oct. 2018. URL: https://solmaz.io/2018/10/18/anatomy-block-stuffing/.

[124] OpenZeppelin. *OpenZeppelin/openzeppelin-solidity*. June 2018. URL: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol.

[125] Daniel Palmer. *SpankChain Loses $40K in Hack Due to Smart Contract Bug*. Oct. 2018. URL: https://www.coindesk.com/spankchain-loses-40k-in-hack-due-to-smart-contract-bug.

[126] PeckShield - batchOverflow Bug. *ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299)*. Apr. 2018. URL: https://blog.peckshield.com/2018/04/22/batchOverflow/.

[127] PeckShield - proxyOverflow Bug. *New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376)*. Apr. 2018. URL: https://blog.peckshield.com/2018/04/25/proxyOverflow/.

[128] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. “T-Fuzz: fuzzing by program transformation”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 697–710.

[129] Christian Reitwiessner. *Formal Verification for Solidity Contracts*. June 2018. URL: https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts.

[130] Josep Sanjuas. *An analysis of a couple Ethereum honeypot contracts*. Dec. 2018. URL: https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum-honeypot-contracts-5c07c95b0a8d.

[131] Alex Sherbachev. *Hacking the Hackers: Honeypots on Ethereum Network*. Dec. 2018. URL: https://hackernoon.com/hacking-the-hackers-honeypots-on-ethereum-network-5baa35a13577.

[132] Alex Sherbuck. *Dissecting an Ethereum Honey Pot*. Dec. 2018. URL: https://medium.com/coinmonks/dissecting-an-ethereum-honey-pot-7102d7def5e0.

[133] Solidity. *Solidity 0.4.24 documentation*. June 2018. URL: http://solidity.readthedocs.io/en/v0.4.24/.

[134] A Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. “Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Vulnerabilities”. In: *arXiv preprint arXiv:1811.06632* (2018).

[135] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. “Smartcheck: Static analysis of ethereum smart contracts”. In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 2018, pp. 9–16.

[136] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. “Securify: Practical security analysis of smart contracts”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 67–82.

[137] Mathy Vanhoef and Frank Piessens. “Symbolic Execution of Security Protocol Implementations: Handling Cryptographic Primitives”. In: *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, 2018. URL: https://www.usenix.org/conference/woot18/presentation/vanhoef.

[138] Marie Vasek and Tyler Moore. “Analyzing the Bitcoin Ponzi scheme ecosystem”. In: *Bitcoin Workshop*. 2018.

[139] Gerhard Wagner. *Smart contract honeypots*. Apr. 2018. URL: https://github.com/thec00n/smart-contract-honeypots.

[140] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. “Erays: Reverse Engineering Ethereum’s Opaque Smart Contracts”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1371–1385.

[141] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. “Fudge: fuzz driver generation at scale”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 975–985.

[142] Jordi Baylina. *Verification of the balances rescued from the multisig compromise*. Dec. 2019. URL: https://github.com/Giveth/WHGBalanceVerification.

[143] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. “Tesseract: Real-time cryptocurrency exchange using trusted hardware”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1521–1538.

[144] ChainSecurity. *Constantinople enables new Reentrancy Attack*. Jan. 2019. URL: https://medium.com/chainsecurity/constantinople-enables-new-reentrancy-attack-ace4088297d9.

[145] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. “A survey on ethereum systems security: Vulnerabilities, attacks and defenses”. In: *arXiv preprint arXiv:1908.04507* (2019).

[146] Peng Chen, Jianzhong Liu, and Hao Chen. “Matryoshka: fuzzing deeply nested branches”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 499–513.

[147] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. “Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 515–530.

[148] ConsenSys Diligence. *Uniswap Audit*. Sept. 2019. URL: https://github.com/ConsenSys/Uniswap-audit-report-2018-12.

[149] MITRE Corporation. *2019 CWE Top 25 Most Dangerous Software Errors*. 2019. URL: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[150] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. “Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges”. In: *arXiv preprint arXiv:1904.05234* (2019).

[151] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. “SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2019, pp. 170–189.

[152] Ethereum Foundation. *Py-EVM - A Python implementation of the Ethereum Virtual Machine*. Aug. 2019. URL: https://github.com/ethereum/py-evm.

[153] Etherscan. *Lendf.Me - MoneyMarket*. 2019. URL: https://etherscan.io/address/0x0eee3e3828a45f7601d5f54bf49bb01d1a9df5ea#code.

[154] Etherscan. *Uniswap: imBTC*. 2019. URL: https://etherscan.io/address/0xffcf45b540e6c9f094ae656d2e34ad11cdfdb187#code.

[155] Josselin Feist, Gustavo Grieco, and Alex Groce. “Slither: a static analysis framework for smart contracts”. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE. 2019, pp. 8–15.

[156] Yu Feng, Emina Torlak, and Rastislav Bodik. “Precise attack synthesis for smart contracts”. In: *arXiv preprint arXiv:1902.06067* (2019).

[159] Aaron Hankin. *Bitcoin Pizza Day: Celebrating the $80 Million Pizza Order*. June 2019. URL: https://www.investopedia.com/news/bitcoin-pizza-day-celebrating-20-million-pizza-order/.

[161] Kenny L. *The Blockchain Scalability Problem & the Race for Visa-Like Transaction Speed*. 2019. URL: https://towardsdatascience.com/the-blockchain-scalability-problem-the-race-for-visa-like-transaction-speed-5cce48f9d44.

[162] Medvedev, Evgeny. *Ethereum ETL v1.3.0*. Apr. 2019. URL: https://github.com/blockchain-etl/ethereum-etl.

[163] Muhammad I. Mehar, Charles L. Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. "Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack". In: *J. Cases Inf. Technol.* 21.1 (2019), pp. 19–32.

[164] OpenZeppelin. *Exploiting Uniswap: from reentrancy to actual profit*. July 2019. URL: https://blog.openzeppelin.com/exploiting-uniswap-from-reentrancy-to-actual-profit/.

[165] Michael Rodler, Wenting Li, Ghassan Karame, and Lucas Davi. "Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*. 2019.

[166] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. *Re-Entrancy Attack Patterns*. July 2019. URL: https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns.

[167] Sergii Kravchenko. *Uniswap audit*. Apr. 2019. URL: https://medium.com/consensys-diligence/uniswap-audit-b90335ac007.

[168] Xin Sun, Quanlong Wang, Piotr Kulicki, and Mirek Sopek. "A simple voting protocol on quantum blockchain". In: *International Journal of Theoretical Physics* 58.1 (2019), pp. 275–281.

[169] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. "Vultron: catching vulnerable smart contracts once and for all". In: *Proc. 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE NIER'19)*. Ed. by Anita Sarma and Leonardo Murta. IEEE, 2019, pp. 1–4.

[170] Yuepeng Wang, Shuvendu Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. "Formal Specification and Verification of Smart Contracts for Azure Blockchain". Apr. 2019. URL: https://www.microsoft.com/en-us/research/publication/formal-specification-and-verification-of-smart-contracts-for-azure-blockchain.

[171] Valentin Wüstholz and Maria Christakis. "Harvey: A greybox fuzzer for smart contracts". In: *arXiv preprint arXiv:1905.06944* (2019).

[172] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. “SolidityCheck : Quickly Detecting Smart Contract Problems Through Regular Expressions”. In: (2019). arXiv: 1911.09425. URL: http://arxiv.org/abs/1911.09425.

[173] William Zhang, Sebastian Banescu, Leonardo Pasos, Steven Stewart, and Vijay Ganesh. “MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract”. In: *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2019, pp. 456–462.

[174] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. “Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities”. In: *Proceedings of the 41st Conference on Programming Language Design and Implementation* PLDI (2020).

[175] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. “SODA: A Generic Online Detection Framework for Smart Contracts”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS’20)*. 2020.

[176] Crytic. *Echidna: Ethereum fuzz testing framework*. Feb. 2020. URL: https://github.com/crytic/echidna.

[177] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. “Empirical review of automated analysis tools on 47,587 Ethereum smart contracts”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 530–541.

[178] Etherscan. *Contracts With verified source codes only*. Nov. 2020. URL: https://etherscan.io/contractsVerified.

[179] Etherscan. *Ethereum Daily Transactions Chart*. Sept. 2020. URL: https://etherscan.io/chart/tx.

[181] Joel Frank, Cornelius Aschermann, and Thorsten Holz. “ETHBMC: A Bounded Model Checker for Smart Contracts”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020.

[182] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. “FuzzGen: Automatic Fuzzer Generation”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020.

[183] J. Little. *pyevmasm – Ethereum Virtual Machine (EVM) disassembler and assembler*. May 2020. URL: https://github.com/crytic/pyevmasm.

[184] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. “Order-fairness for byzantine consensus”. In: *Annual International Cryptology Conference*. Springer. 2020, pp. 451–480.

[185] John Lilic. *EIP-2878 – Block Reward Reduction to 0.5 ETH*. 2020. URL: https://github.com/ethereum/EIPs/pull/2878.

[186] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. "sFuzz: an efficient adaptive fuzzer for solidity smart contracts". In: *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020, pp. 778–788.

[187] PeckShield. *Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis*. Apr. 2020. URL: https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[188] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. "Verx: Safety verification of smart contracts". In: *Proc. 41st IEEE Symposium on Security and Privacy (IEEE SP'20)*. IEEE, 2020, pp. 18–20.

[189] SWC Registry. *Smart Contract Weakness Classification and Test Cases*. Nov. 2020. URL: https://swcregistry.io/.

[190] Christian Reitwiessner. *Solidity 0.6.6 Documentation: Layout of State Variables in Storage*. July 2020. URL: https://solidity.readthedocs.io/en/v0.6.6/miscellaneous.html#layout-of-state-variables-in-storage.

[191] Duncan Riley. *$25M in cryptocurrency stolen in hack of Lendf.me and Uniswap*. Apr. 2020. URL: https://siliconangle.com/2020/04/19/25m-cryptocurrency-stolen-hack-lendf-uniswap/.

[192] Ryan Sean Adams. Sept. 2020. URL: https://twitter.com/RyanSAdams/status/1252574107159408640.

[193] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. "eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts". In: *arXiv preprint arXiv:2005.06227* (2020).

[194] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. "VeriSmart: A highly precise safety verifier for Ethereum smart contracts". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1678–1694.

[195] Péter Szilágyi. *Go-Ethereum Management APIs - JavaScript-based tracing*. Jan. 2020. URL: https://github.com/ethereum/go-ethereum/wiki/Management-APIs%5C#javascript-based-tracing.

[196] UDE Secure Software System Research Group. *EVMPatch Evaluation Data*. Dec. 2020. URL: https://github.com/uni-due-syssec/evmpatch-eval-data.

[197] Dabao Wang, Siwei Wu, Ziling Lin, Lei Wu, Xingliang Yuan, Yajin Zhou, Haoyu Wang, and Kui Ren. "Towards Understanding Flash Loan and its Applications in DeFi Ecosystem". In: *arXiv preprint arXiv:2010.12252* (2020).

[198] Lei Wu, Siwei Wu, Yajin Zhou, Runhuai Li, Zhi Wang, Xiapu Luo, Cong Wang, and Kui Ren. “EthScope: A Transaction-centric Security Analytics Framework to Detect Malicious Smart Contracts on Ethereum”. In: *arXiv preprint arXiv:2005.08278* (2020).

[199] Xiao Liang Yu, Omar I. Al-Bataineh, David Lo, and Abhik Roychoudhury. “Smart Contract Repair”. In: *ACM Trans. Softw. Eng. Methodol.* 29.4 (2020), 27:1–27:32.

[200] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. “TXSPECTOR: Uncovering Attacks in Ethereum from Transactions”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2775–2792. ISBN: 978-1-939133-17-5.

[201] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. “EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts”. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 116–126.

[202] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. “Smartshield: Automatic smart contract protection made easy”. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 23–34.

[203] Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao, Zhemin Yang, and Yuan Zhang. “An Ever-evolving Game: Evaluation of Real-world Attacks and Defenses in Ethereum Ecosystem”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2793–2810. ISBN: 978-1-939133-17-5.

[204] C. Torres. *Horus: A framework to detect attacks and trace stolen assets across Ethereum (FC 2021)*. Jan. 2021. URL: https://github.com/christoftorres/Horus.

[205] Catalin Cimpanu. *Hackers steal $130 million from Cream Finance; the company’s 3rd hack this year*. Oct. 2021. URL: https://therecord.media/hackers-steal-130-million-from-cream-finance-the-companys-3rd-hack-this-year/.

[206] ChainSecurity. *ChainSecurity*. 2021. URL: https://chainsecurity.com/.

[207] CoinMarketCap. *Top 100 Crypto Coins by Market Capitalization*. Dec. 2021. URL: https://coinmarketcap.com/coins/.

[208] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. “EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode”. In: *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 2021, pp. 127–137.

[212] William Foxley. *Ethereum Miners Earned Record $830M in January*. 2021. URL: https://www.coindesk.com/ethereum-miners-earned-record-830m-january.

[213] J. Feist. *EVM CFG BUILDER – EVM CFG recovery*. Mar. 2021. URL: https://github.com/crytic/evm_cfg_builder.

[214] Tai D. Nguyen, Long H. Pham, and Jun Sun. "SGUARD: Towards Fixing Vulnerable Smart Contracts Automatically". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1215–1229.

[215] OpenZeppelin. *OpenZeppelin*. 2021. URL: https://openzeppelin.com/.

[216] OpenZeppelin Docs. *Writing Upgradeable Contracts*. Aug. 2021. URL: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable.

[217] PeckShield. *PeckShield - Industry Leading Blockchain Security Company*. 2021. URL: https://peckshield.com/.

[218] Daniel Perez and Benjamin Livshits. "Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited". In: *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021.

[219] Michael Rodler, Wenting Li, Ghassan Karame, and Lucas Davi. "EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts". In: *30th USENIX Security Symposium (USENIX Security '21) [To be published]*. Vancouver, B.C.: USENIX Association, Aug. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/rodler.

[220] SmartBugs. *SmartBugs: A Framework to Analyze Solidity Smart Contracts*. Aug. 2021. URL: https://github.com/smartbugs/smartbugs.

[221] Sandeep Soni. *Ethereum now more valuable than Visa, JPMorgan Chase; Bitcoin-rival among top five financial services*. May 2021. URL: https://www.financialexpress.com/market/ethereum-now-more-valuable-than-visa-jpmorgan-chase-bitcoin-rival-among-top-five-financial-services/2247384/.

[223] Vyper. *Pythonic Smart Contract Language for the EVM*. Dec. 2021. URL: https://github.com/ethereum/vyper.

[224] Sam M Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J Knottenbelt. "SoK: Decentralized Finance (DeFi)". In: *arXiv preprint arXiv:2101.08778* (2021).

[225] Gavin Zheng, Longxiang Gao, Liqun Huang, and Jian Guan. "Upgradable Contract". In: *Ethereum Smart Contract Development in Solidity*. Springer, 2021, pp. 197–213.

[227] Robinson Dan and Konstantopoulos Georgios. *Ethereum is a Dark Forest*. URL: https://medium.com/@danrobinson/ethereum-is-a-dark-forest-ecc5f0505dff.

[228] William Foxley. *DeFi Has a Front-Running Problem. Sparkpool's Potential Fix Is Launching This Month*. URL: https://www.coindesk.com/sparkpool-taichi-mining-network-front-running-defi.

[229] Alex Manuskin. *Ethology: A Safari Tour in Ethereum's Dark Forest*. URL: https://zengo.com/ethology-a-safari-tour-in-ethereums-dark-forest.

[230] Naz. *How to Front-run in Ethreum*. URL: https://nazariyv.medium.com/crypto-front-running-for-dummies-bed2d4682db0.

[231] Jonathan Otto. *Arbitraging Uniswap and SushiSwap in Node.js*. URL: https://messari.io/article/arbitraging-uniswap-and-sushiswap-in-node-js.

[232] SAMCZSUN. *Escaping the Dark Forest*. URL: https://samczsun.com/escaping-the-dark-forest.

[233] Zack Voell. *Decentralized Exchange Volumes Up 70 percent in June, Pass 1.5B*. URL: https://www.coindesk.com/decentralized-exchange-volumes-up-70-in-june-pass-1-5-billion.