FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informatik

Security Investigation in Encrypted Environment

Fatih Kilic

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. Georg Carle

Prüfer der Dissertation:

1. Prof. Dr. Claudia Eckert
2. Prof. Dr. Felix Freiling

Die Dissertation wurde am 03.07.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.10.2017 angenommen.

Danksagungen

An dieser Stelle möchte ich mich bei allen wunderbaren Menschen bedanken, die mich während meiner Zeit als Doktorand unterstützt haben.

Ein großer Dank geht an Frau Prof. Dr. Claudia Eckert, die mir die Möglichkeit zur Promotion angeboten hat und mich auch über die Jahre immer tatkräftigt unterstützt hat. Ohne Dich gebe es diese Dissertation nicht, vielen Dank Claudia. Weiterhin bedanke ich mich bei Herr Prof. Dr. Felix Freiling, der meine Arbeit unterstützt und als Zweitgutachter bewertet hat.

Als nächstes möchte ich mich bei meinen aktuellen und ehemaligen Kollegen bedanken, mit denen ich zusammenarbeiten durfte und sehr interessante Gespräche führen durfte. Das sind meine Kollegen vom Fraunhofer-Institut für Sichere Informationstechnologie (SIT) in Darmstadt, von der Technische Universität München (TUM) am Lehrstuhl für IT-Sicherheit und zuletzt vom Fraunhofer-Institut für Angewandte und Integrierte Sicherheit (AISEC). Namentlich möchte ich jedoch folgende Personen in alphabetischer Reihenfolge hervorheben. Dr. Christian Schneider, Prof. Dr. Georg Sigl, Dr. Hasan Ibne Akram, Dr. Sebastian Vogl und Dr. Thomas Kittel. Darüber hinaus bedanke ich mich auch bei den ehemaligen und aktuellen Studenten, die in dieser Zeit Beiträge zu meinen Projekten geleistet haben. Gesonderter Dank gilt hierbei Sergej Proskurin, Hannes Laner, Benedikt Geßele und Richard von Seck.

Mein besonderer Dank gilt meiner Familie. Vielen Dank an meine Eltern, die mich seit meiner Geburt immer motiviert und unterstützt haben, sowohl finanziell als auch mental. Vielen Dank an meine Brüder Baybars und Yalcin, die jederzeit für mich da waren. Vielen Dank auch an meine Schwägerin, die über die Zeit mich mit leckerem Essen versorgt hat. Ohne die Liebe, die Unterstützung und die Ermutigung meiner Familie wäre dieser Weg nicht möglich gewesen.

Abstract

Nowadays, a wide variety of applications are using encryption to protect their confidential data in network communication. Since currently available security tools are mainly developed for plaintext network transmission, encryption obstructs security analysis and protection.

In this thesis, we elaborate on the challenges resulting from encrypted network communication. In fact, encryption does not stop intruders from exploiting application vulnerabilities. However several protection mechanisms, e.g. kernel-based, compiler-based or third-party libraries, help to mitigate the success of attacks. We examine the feasibility of attacks against vulnerable applications on modern systems to identify the need of additional protection mechanisms. One approach is the identification of the vulnerability in order to fix it. Security testing can be applied to analyse applications with common attack payloads. For applications using encryption, the analyst has to discover the encryption algorithm and the appropriate key first, in order to proceed with testing. Another approach is to inspect the network traffic for known attack signatures. However, inspecting network data of applications with an active end-to-end encryption is not feasible on the network level without increasing the attack surface. Host-based solutions can help here, but they are limited in their use cases and prone to local attacks. Analysing binary applications without having access to the source code is known as reverse engineering. The identification of specific locations inside the application is a very labour-intensive process, but necessary for further analysis.

The contributions of our work are as follows. First, we propose a new method for exploiting vulnerabilities over the network to show that system-based or application-based protection mechanisms are not sufficient. Second, we provide a framework for analysing binaries using encrypted network communication, sustaining the end-to-end encryption. The developed modules of the framework allow us to intercept, extract, modify and inject plaintext data, which is transmitted encrypted over the network. We propose a generic method to analyse applications with encrypted network communication without breaking the end-to-end encryption. Using this framework, we create a data bridge for

security testing applications, to reduce the problem of testing with encrypted protocols to testing with plaintext protocols. To protect network applications from intruders, we build a second data bridge for virtual machine introspection. We use virtualization extensions to isolate our generic solution. This allows to protect monitoring/inspection tools, e.g. intrusion detection systems, against further attacks. In our work we rely on specific functions inside applications. Finally, we present a second framework for efficient function identification in binary applications. We propose a method to speed up the localization of specific functions. Our dynamic approach reduces the number of executed functions significantly and therefore the time of the analysis process. To evaluate the effectiveness of our framework for encrypted network communication, we implemented a prototype for the x86 and ARM architecture. The evaluation shows that our solution is equipped for attacks against the framework.

Zusammenfassung

Verschlüsselung wird gegenwärtig von einer Vielzahl von Anwendungen genutzt, um vertrauliche Daten in der Netzwerkkommunikation zu schützen. Da die derzeit verfügbaren Sicherheitsapplikationen hauptsächlich für Klartextübertragungen entwickelt wurden, beeinträchtigt die Verschlüsselung die Sicherheitsanalyse und den Schutz der Applikation.

In der vorliegenden Arbeit werden die aus der verschlüsselten Netzwerkkommunikation resultierenden Herausforderungen untersucht. Verschlüsselung hält Angreifer nicht davon ab, vorhandene Schwachstellen in Anwendungen auszunutzen. Gleichwohl existieren mehrere Schutzmechanismen, die z.B. im Kernel und Übersetzer oder in Bibliotheken von Drittanbietern integriert sind. Diese schränken die Erfolgswahrscheinlichkeit der Angriffe ein. Vor diesem Hintergrund wird die Durchführbarkeit von Angriffen bei anfälligen Anwendungen auf modernen Systemen untersucht, um die Notwendigkeit zusätzlicher Schutzmechanismen zu ermitteln. Ein möglicher Ansatz könnte die Identifizierung der Schwachstelle sein, um diese zu beheben. Durch Sicherheitstests kann die Anwendung mittels bekannter Angriffsdaten analysiert werden. Für Anwendungen, die Verschlüsselung verwenden, muss der Analyst den Verschlüsselungsalgorithmus und den dazugehörigen Schlüssel zuerst ermitteln, um sodann mit dem Testen fortfahren zu können. Ein weiterer Ansatz besteht darin, den Netzwerkverkehr auf bekannte Angriffssignaturen zu untersuchen. Jedoch ist die Inspektion von Applikations-Netzwerkdaten mit aktiver Ende-zu-Ende-Verschlüsselung auf Netzwerkebene nicht möglich, ohne zusätzliche Angriffsvektoren hinzuzufügen. Host-basierte Lösungen können hier helfen, aber ihre Anwendungsmöglichkeiten sind begrenzt und außerdem anfällig für lokale Angriffe. Die Analyse der Binäranwendung ohne den Quellcode zu besitzen, wird als Reverse Engineering bezeichnet. Die Identifizierung spezifischer Bereiche innerhalb der Applikation ist ein sehr arbeitsintensiver Prozess, der jedoch für eine weiterführende Analyse unverzichtbar ist.

In der vorliegenden Arbeit wird zunächst eine neue Methode zur Ausnutzung von Schwachstellen über das Netzwerk vorgestellt, um zu verdeutlichen, dass ein system- oder an-

wendungsbasierter Schutzmechanismus nicht ausreicht. In einem zweiten Schritt wird ein Rahmenwerk für die Analyse von Binärdateien mit verschlüsselter Netzwerkkommunikation präsentiert, die dabei die Ende-zu-Ende-Verschlüsselung beibehält. Die entwickelten Module des Rahmenwerks erlauben es, die Klartextdaten, die verschlüsselt über das Netzwerk übertragen werden, abzufangen, zu extrahieren, zu modifizieren und zu injizieren. Es wird eine generische Methode vorgeschlagen, um die Anwendung mit verschlüsselter Netzwerkkommunikation zu analysieren, ohne die Ende-zu-Ende-Verschlüsselung aufzubrechen. Mit diesem Framework wird eine Datenbrücke für Sicherheitstestanwendungen erstellt, um das Problem des Testens mit verschlüsselten Protokollen auf das Testen mit Klartextprotokollen zu reduzieren. Um Netzwerkanwendungen vor Eindringlingen zu schützen, wird eine zweite Datenbrücke für die Überwachung der virtuellen Maschine zusätzlich hinzugefügt. Es werden Virtualisierungserweiterungen genutzt, um die generische Lösung zu isolieren. Dies schützt Überwachungs- / Inspektionswerkzeuge, wie z.B. Intrusion Detection Systeme, gegen weiterführende Angriffe. In der vorliegenden Arbeit werden konkrete Funktionen innerhalb der Anwendungen benötigt. Abschließend wird ein zweites Rahmenwerk für eine effiziente Funktionsidentifikation in binären Anwendungen vorgestellt. Es wird eine Methode entwickelt, um die Lokalisierung bestimmter Funktionen zu beschleunigen. Der neue dynamische Ansatz reduziert die Anzahl der zu analysierenden ausgeführten Funktionen deutlich und damit auch die Zeit des Analyseprozesses. Um die Funktionsfähigkeit des Rahmenwerks für verschlüsselte Netzwerkkommunikation zu zeigen, wurde eine Prototyp auf der x86- und ARM-Architektur implementiert. Die Auswertung zeigt, dass die entwickelte Lösung für Angriffe gegen das Rahmenwerk vorbereitet ist.

Contents

List of Publications

[1] Fatih Kilic, Thomas Kittel, and Claudia Eckert. "Blind Format String Attacks". In: *Proceedings of the 10th International Conference on Security and Privacy in Communication Networks (SecureComm 2014)*. Vol. 153. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer International Publishing, 2015, pp. 301–314.

[2] Fatih Kilic, Hannes Laner, and Claudia Eckert. "Interactive Function Identification Decreasing the Effort of Reverse Engineering". In: *Proceedings of the 11th International Conference on Information Security and Cryptology (Inscrypt 2015)*. Springer International Publishing, 2016, pp. 468–487.

[3] Fatih Kilic and Claudia Eckert. "iDeFEND: Intrusion Detection Framework for Encrypted Network Data". In: *Proceedings of the 14th International Conference on Cryptology and Network Security (CANS 2015)*. Vol. 9476. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 111–118.

[4] Fatih Kilic, Benedikt Geßele, and Hasan Ibne Akram. "Security Testing over Encrypted Channels on the ARM Platform". In: *Proceedings of the 12th International Conference on Internet Monitoring and Protection (ICIMP 2017)*. 2017.

[5] Sergej Proskurin, Fatih Kilic, and Claudia Eckert. "Retrospective Protection utilizing Binary Rewriting". In: *14. Deutscher IT-Sicherheitskongress*. May 2015.

[6] Sebastian Vogl, Fatih Kilic, Christian Schneider, and Claudia Eckert. "X-TIER: Kernel Module Injection". In: *Proceedings of the 7th International Conference on Network and System Security*. Vol. 7873. Lecture Notes in Computer Science. Springer, June 2013, pp. 192–206.

Chapter 1

Introduction

Nowadays, the internet is growing rapidly and undergoing big changes. On one hand the number of devices connected to the internet grows and the range of types of devices increases. On the other hand, in the last few years, the concept of Internet Of Things (IoT) has emerged and evolved rapidly to bring to the connected world not only desktop computers, mainframes and mobile devices, but also all types of other devices such as cars, medical devices, various sensors and smart home devices. As a consequence to these changes, the attack surface for cybercrime is not only getting larger, but also more diversified.

Therefore, the usage of encryption in companies increased over the last years. According to a recent survey of thales security[38], the current extensive usage of encryption has an average deployment rate of 41%. At the same time the usage of encryption is also enabling a new attack surface for companies. Based on the study of A10 and the Ponemon institute[1] in 2016, about 80% of organisations were attacked by cyber-criminals. About half of these attacks were hidden in encrypted traffic to evade detection. 75% of the participants admitted, that the intruder was able to steal employees' credentials from their network. The majority of the respondents (62%) does not decrypt and subsequently inspect the encrypted traffic. The interesting point is, that 47% of them see the reason in the lack of good security tools.

Recent statistics about cyber-attacks show, that the number of unknown and targeted attacks is increasing[53]. An example for an unknown or a targeted attack, could be the exploitation of a vulnerability inside an application, which is using encrypted network communication. If no package inspection can be performed on the network traffic, the system has to protect or mitigate the attacks against the application. Buffer Overflow and Format String Vulnerabilities are two well known examples of this. Although these attacks are known for many years, there is still a number of applications that have been found to be vulnerable to such attacks in the recent years. Thus we can assume, that

this type of vulnerabilities will still be present in future. Currently, there are compiler-based or system-based protection mechanisms to restrict the successful exploitation of these kind of vulnerabilities. It has to be investigated if these protections are sufficient to circumvent an attack in all cases without the inspection of network data. Especially for the case of Format String Attacks, which are based on precise rules on how to exploit a vulnerability, e.g. the usage of specific format string specifiers. With this knowledge, generic rules for an Intrusion Detection System can be generated, to detect Format String Attacks by inspecting the network data.

The omnipresent threat of network intrusions underlines that the necessity for effective intrusion detection systems are indispensable in modern infrastructures. Regular intrusion incidents emphasize that protection against network attacks remains a great challenge. While state-of-the-art Network Intrusion Detection and Analysis Systems focus on suspicious traffic, they lack an effective analysis of end-to-end encrypted communication activity. Contemporary Host-based Intrusion Detection Systems complement traffic analysis by an additional fine-grained investigation on the host, yet, they are prone to local attacks.

The security of a system should not rely only on the presence of a protection system. To avoid successful attacks, the vulnerabilities of the system should be identified before the intruder acts. A common way that has been applied for many years to detect vulnerabilities in applications consists in security testing. With the increasing demand for encryption to protect the confidentiality of network data, the requirements have changed. When proprietary, closed-source software uses end-to-end encryption, security testing tools, which are testing the application layer over network with plaintext data, will eventually fail.

Analysing binary applications to identify vulnerabilities requires reverse engineering techniques. Today's software is growing in size and complexity. Consequently, analysing closed-source binaries becomes time-consuming and labour-intensive. In the common use case, the analyst is only interested in specific functions of the given application. However, identifying the relevant functions is difficult since no related meta information is given.

In this thesis, we will focus on the problems coming with encrypted network communication and elaborate an applicable solution.

First, we investigate if software vulnerabilities are still exploitable with modern protection mechanisms. Therefore, we evaluate the applicability of Format String Attacks with active compiler-based and system-based protection mechanisms. Attacks from this category are easy to detect using network data analysis, if there is a possible way to inspect the plain text of the network data. Since attacks over encrypted network data are still a problem for many companies, we selected Format String Attacks as an example

to proof that we should not rely only on compiler-based and system-based protection mechanisms.

Second, we present a framework for inspecting encrypted network data without breaking the security model of end-to-end encryption. Our approach does not require any source code of the involved applications and thereby also protects closed source applications. Our framework works independently of the utilized encryption key. We present two use cases how our framework can detect intruders by analysing the network data and how we can test remote applications with enabled network data encryption. To achieve this, the framework detects the relevant functions in the target application, extracts and subsequently inspects the data. In order to test remote applications, the framework intercepts and injects user controlled data into the application. We also transfer the framework to the ARM architecture and thereby, make it suitable for embedded devices.

Additionally, we evaluate our framework against dedicated attacks to bypass or deactivate it. To circumvent these attacks, we enhance our framework utilizing virtualization technology. This allows us to support Host-based Intrusion Detection Systems in a live environment. As such, we create an isolated analysis system that is resilient to local attacks. More precisely, we employ Virtual Machine Introspection to effectively utilize and protect hardware breakpoints from the guest to perform exhaustive traffic analysis without disturbing the overall guest operation.

The methods used in the framework require reverse-engineering techniques, especially if the target is a closed-source application. The identification of specific functions inside the binary is necessary to enable the data bridge between the target, which has to be protected or analyzed, and the framework itself. If the automated process fails, an analyst has to analyse the binary manually to detect the required functions. To speed up this reverse-engineering process and to provide a more generic solution for function detection, we propose another framework in this thesis. We use the benefits of Dynamic Binary Instrumentation as a base to collect executed function calls. We support the analyst in filtering the relevant functions for specific functionality. Our approach is divided into three steps. Namely real-time data gathering, user defined information processing/filtering and graphical representation. We show a significant speed-up in the reverse engineering process using our framework. In fact, we reduce the number of executed functions to be viewed by the analyst. Furthermore by means of the visual components, the analyst pre-selects the functions on an abstract level.

In the following Section 1.1 we describe the problem states, in Section 1.2 we summarize the contributions in detail and give a brief outline in Section 1.3.

1.1 Problem Statement

In this thesis we face different challenges to achieve the overall solution for security investigation in encrypted environments. First, we have to analyse if modern system-based and compiler-based protection mechanisms are sufficient to mitigate attacks or if additional security tools are necessary to protect vulnerable network applications. Second, we have to identify a solution to be able to adapt current solutions, like network security testing tools or intrusion detection systems, to operate with encrypted network communication. Finally, we have to elaborate an efficient way for the localisation of certain functions inside binary applications.

Exploitation of Vulnerable Applications over the Network

Over the years many protection mechanisms against exploitation of network based applications have been developed and implemented. They prevent the successful exploitation of vulnerabilities from attacks over the network. One protection feature is disabling the execution of code in writeable memory regions. This prevents the execution of shellcode in user controlled buffers. A second feature is to enable address space layout randomization. This makes it difficult for an attacker to identify the exact address in remote memory. There are many other protections like this to mitigate remote exploitation. The challenge here is to create a generic method to show that even with active protection mechanisms known vulnerabilities can still be exploited on modern systems.

Security Analysis with enabled Encrypted Network Communication

To protect the confidentiality of the payload message in network data, many applications use encryption. If end-to-end encryption is applied, then only the communication participants can view the payload. However, this leads to new challenges in the security world. For example, intrusion detection tools, which rely on inspection of plaintext data will not be able to protect against malicious content. Considering security testing as another example, security analysts cannot use classical testing tools operating on the network level. They need to reverse engineer the encryption key and algorithm to be able to communicate with the remote host. In order to solve all issues together, a generic solution for the encrypted network communication use case, should be found. Since we already have solutions for unencrypted network communication, we only need to build a bridge between the current tools and the encrypting applications and keep up the end-to-end encryption.

Efficient Function Identification in Binaries

Establishing the data bridge can only be done by having an endpoint inside the binary itself before the encryption and after the decryption is executed. This endpoint can be a function inside the target application, which is responsible for the encrypted network communication. On closed-source binaries, the only way to identify these functions is using reverse engineering techniques, which can be very time-consuming. This poses the challenge of identifying the required functions in an efficient way.

1.2 Contributions

In this section, we summarize our contributions.

New Methods for Exploiting Vulnerabilities over the Network

To evaluate the feasibility of exploits on modern systems, we investigated the limitations of the used protection mechanisms. For our analysis we selected format string vulnerabilities as a concrete example. There are two reasons for this decision. First, they are easy to detect if the source code is available, because the number of vulnerable functions is limited and can be automated. Second, the attack has to follow specific exploitation rules, namely concrete format string parameters. This limits the attack payload and cannot vary much and should be detected more easily.

We consider the protection mechanisms of the operating system, compiler and third party libraries to mitigate successful exploitation of security vulnerabilities in the application layer. We provide a new method to show, that network based applications can still be successfully exploited even with active protection mechanism. Our method introduces a novel mechanism that enables an attacker to write to arbitrary memory locations using a format string attack without the requirement to place the format string onto the stack, which is the base requirement of classical attacks. Furthermore we describe a technique to redirect the control flow of a format string attack vulnerable function in a blind fashion. This means, that a memory leakage is not needed anymore.

Development of a Framework for Security Analysis

Knowing the fact, that general mitigation techniques are not enough to protect against application vulnerabilities like format string attacks, we need solutions to analyse binaries even with enabled encrypted network communication. The main challenge is to analyse the network application while keeping up the end-to-end encryption. The payload

of the network transmission is encrypted and decrypted within the responsible functions inside the application. We use reverse engineering techniques to access this payload in the application memory. To achieve this, we need to identify the memory location of the payload and the state of the application, when the payload is available.

Therefore, we provide an analysis framework, called iDeFEND, for closed source binaries, that are communicating over the network using an encrypted channel. Using our framework we are able to analyse the payload of the network data out of the application without breaking the security of end-to-end encryption.

Data Bridge for Security Testing

Without the knowledge of the encryption algorithm and key, remote testing of network-based applications with enabled encryption on the network channel becomes a challenging task. In our approach, we provide an interface for the analyst to analyse and inject plaintext data in the network communication. This way, we reduce the problem of testing with encrypted protocols to the testing with plaintext protocols and thus, enable the usage of many already existing testing tools.

To achieve this we provide a method to identify network security related functions in applications. As an example, these functions are responsible for the plaintext parsing, encryption and sending. By taking control over these functions, we are able to inject, intercept, extract and modify the payload in plaintext. This allows us to analyse applications using encrypted traffic without the need for reverse engineering of the encryption algorithm and key.

Data Bridge for Virtual Machine Introspection

Being able to inspect the payload of network data helps to create a data bridge for intrusion detection systems. However, modifying the target application or running the framework on the same operating system creates new attack vectors. The problem of host-based intrusion detection systems is, that they are vulnerable after a successful attack gaining root privileges. The same problem affects our framework, if we want to bridge the data for the use case of intrusion detection.

To solve this issue, we use the isolation concepts of virtualization to be separated from the operating system of the monitoring target. More precisely we extract decrypted network data for analysis through hypervisor functionality, using hardware breakpoints. To achieve this, we handle hardware breakpoints using virtual machine introspection techniques. With this approach we guarantee the breakpoint execution within the guest.

Furthermore we hide hardware breakpoints from the target system and protect our framework against deactivation.

Identification of Functions inside Binary Applications

The framework relies on the identification of designated functions inside closed-source applications. For the case that the automated approach fails, an analyst has to reverse engineer the required functions manually. Additionally, the function detection approach of iDeFEND is dedicated to the use case of encrypted network communiciation. To cover other use cases, e.g. identification of database relevant functions, we need a more generic approach. Since manual reverse engineering and identifying functions is time-consuming and labour-intensive, we need efficient methods.

We propose a new method to decrease the time overhead using dynamic binary instrumentation. In detail we log function calls of a target application. Our approach enriches the data by providing labelling of program states. We process the gathered data using common set operations. Finally, we illustrate the processed information with highlighting. This will reduce the amount of functions significantly.

Demonstration of the Effectiveness of our Framework

We have implemented the described Framework, which is called iDeFEND, on the x86 and ARM architecture. We enhanced it using virtualization extensions. With several experiments, we tested the applicability of our framework. The achieved results are showing that our framework is able to inspect, intercept, modify, and inject the plaintext data inside the encrypted network channel. Finally, we evaluated the framework against common attacks and showed the effectiveness of iDeFEND.

1.3 Outline

The structure of this thesis is as follow.
In Chapter 2, we provide background information as it is needed to understand the details of this thesis. Since this thesis is combining different research areas, we are explaining the related information of each area. Since our implementations are depending on different CPU architectures, we describe the required background of the x86 and ARM architectures. Furthermore, we give an overview about dynamic binary instrumentation, which is used to identify the required functions in our work.

In Chapter 3, we discuss the related work.

In Chapter 4, we show that vulnerabilities on application level are still a big threat by introducing a new exploitation method. This technique can be used to exploit a state of the art full patch system over the encrypted network channel without being detected by a network intrusion detection system.

In Chapter 5, we provide the concept of a framework to address the problematic of encrypted network traffic. This chapter shows how to bypass the problem for closed source application without knowing the encryption key or the algorithm.

In Chapter 6, we discuss how to do blackbox security tests for closed source application which are using encrypted network communication. Furthermore, we also explain the difference for the ARM platform to test mobile applications.

Chapter 7 expands the security of the framework by moving it into the hypervisor level to achieve protection and reliability against successful attacks.

The identification of the required functions will be discussed in Chapter 8.

In Chapter 9, we show the effectiveness of the framework using an overall evaluation. We show the adaptability by describing concrete scenarios and discuss the attacks against our framework.

Finally, we conclude this thesis in Chapter 10.

Chapter 2

Background

This thesis is considering different research topics, therefore we will explain the related background to the reader. The focus of this thesis is on applications using encrypted network communication. We will start with a brief overview end-to-end encryption in network communication. The used methods are relying on CPU architectures, so we will continue describing the relevant information for the x86 and ARM architectures. Next, we continue on the topic of dynamic binary instrumentation, which is used for binary analysis. As last, we give a short overview about the basics of format string attacks, which are used in our work.

2.1 End-To-End Encryption in Network Communication

Encryption has a wide range of applications on computers, smart phones and many other embedded devices that are connected to the internet. Communicating over the internet means communicating over a huge network where all devices are connected with each other. In such large networks it is impractical to have direct paths between all devices and hence, the communication between two devices must be routed over several networking nodes, for example servers [39, p. 3]. This of course introduces a confidentiality problem, because all nodes between two communication partners have potential access to the content. Additionally, a device has no real influence on the path in a network. The routing of a communication channel is abstracted and handled by the network layer of the OSI model. Applications, e.g. a messenger on a computer, operate on the application layer of the OSI model in order to send messages between two communication partners. When a user decides to send a text message, this text is wrapped as a payload in a networking packet and is then delivered over several, probably not trustworthy, networking nodes to the destination device. If the messenger

communicates unencrypted, every node in between sender and recipient can read the messages by simply unwrapping the network packets. Encrypting the payload of the network packet prevents other nodes from acquiring information and still guarantees the network protocol to deliver the packet. In general, the confidentiality of the message is only given if the encryption is end-to-end. This means that only sender and recipient can decrypt messages. As soon as another node is involved into the encryption chain, it is not end-to-end encryption anymore and potentially insecure to confidentiality violations. Usually such additional nodes are servers that handle communications of an application or are used to store chat logs from messengers.

Another problem comes with the authenticity in networks. For asymmetric cryptography, a confidential communication can be started by obtaining the public key of the communication partner [35]. However, if a partner uses the public key of an attacker as the key of his actual communication partner, the attacker is part of the confidential communication and is able to decrypt messages and acquire information. Furthermore, the attacker could deceive both communication partners and have full access to the whole communication. This is called Man-In-The-Middle attack since the attacker is communicating with both partners with separate keys and simply forwards all messages [87]. This way, the victims cannot even notice the intruder. The attacker is in a position where he can access all data and even manipulate or inject new messages. Symmetric cryptography also suffers under this attack, since the secret key must be exchanged first, which is generally realized by asymmetric cryptography. A solution to detect attackers, for example, is to have a trusted third party that knows both trusted communication parties and verifies their public keys to each other. Not using end-to-end encryption but multiple chained encryptions in a conversation increases the risk of being successfully deceived by an attacker since the amount of attackable nodes increases.

2.2 CPU Architecture

With the spreading demand for processors in high-performance computers, desktop computers, smart phones and other embedded devices, the processor architectures have to cope with a larger and larger range of applications. Each area of application requires the optimization of different factors like computational power, energy consumption, real-time constraints and physical robustness. As a result, two of the currently market-leading processor architectures ARM and x86 are based on significantly differing designs and thus, dominant different areas of computing. In the following subsections, we present those differences between ARM and x86, regarding their architectural design, application level programming and debugging interface.

2.2.1 The x86 Architecture

One of the currently most distributed designs on the market is the x86 architecture. It is dominating the sector of high performance computing and desktop computers. x86 is designed as a backward compatible 32 bit CISC architecture with a comparatively large instruction set of over one thousand instructions [59]. This makes it especially suited for high performance applications where power consumption is of a lower priority. One of the performance factors is the complex encoding pattern of instructions that allows to efficiently implement a huge amount of instructions. Instructions have variable lengths and consist of operation code bytes which are followed by several prefixes that indicate the actual operations of the processor. Furthermore, many instructions exist that combine several small operations. On x86, for example, instructions take memory addresses as parameters, even as destination of computations. This way, only one instruction is needed to wrap tasks over several processor cycles and allow to better optimize parallel execution. Since our hardware is running an x86 Intel processor, we decided to focus on the IA32 and we will refer to it as x86 in the following chapters.

2.2.1.1 Application Level Registers

A processor in general provides many different registers for a multitude of purposes. For programming on application level, this set typically shrinks down to a few important registers like for example the general purpose registers. On x86 most registers, especially the general purpose registers, are accessible in different sizes. For instance, the general purpose register `EAX` is a 32 bit register where the lower 16 bit can be accessed by referring to register `AX`. `AL` and `AH` again are subsets for the lowest and second lowest 8 bit of `AX`. On x64, there is also a 64 bit variant `RAX` that is a superset of `EAX`. [59]

x86 provides the following set of registers.

- **General Purpose Registers**
  Programmers can use eight different general purpose registers namely `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, `ESP` and their subsets as explained before. The first four registers are freely usable for any kind of purpose and are accessible in 32, 16 and 8 bit versions. The four others are only provided in 32 and 16 bit variants. Caution should be used especially on the latter two, `EBP` and `ESP`, as they are typically representing the current stack frame pointer and the stack pointer itself. On x64 all of these registers are also available in 64 bit.

- **Instruction Pointer**
  The program counter `EIP` always points to the instruction that is executed next. It is not directly accessible, but rather indirectly through jumps or calls [59, p. 80].It

has a subset register for the lower 16 bit and on x64, this register has a 64 bit superset RIP.

- **Program Status and Control Register**
  On x86 arithmetic or logical computations always create information about for example an overflow, zero as the result of an operation and a carry bit. The purpose of the status register is that it holds flags for each information.

- **Segment Registers**
  The segment registers hold segment selectors of 16 bits length. A segment selector is a special pointer that identifies a segment in memory.

2.2.1.2 Breakpoints on x86

The most basic and important feature of debugging is to interrupt a target application at a specified point of execution, in order to analyse and manipulate its state. Therefore, each processor architecture introduces different types of breakpoints and hardware interfaces. Programmers generally can choose between two types of breakpoints: the software and the hardware breakpoint.

- **Software Breakpoint**
  On the hardware level, a software breakpoint is an instruction for which the execution is trapping the processor in a debugging state. Software breakpoints must be placed in the executable code, which are only triggered on execution. They have comparatively much overhead and thereby perform slower. However, they have their advantages in being quantitatively unlimited.

  The x86 architecture implements software breakpoints as interrupts from the interrupt calling instruction *Call to Interrupt Procedure* that takes the number of the to-be-invoked interrupt routine as an operand. The instruction generates two bytes code, one byte for the operation code (short opcode) and one for the number of the interrupt handling routine. Interrupt three, that means instruction `INT3`, has a special implementation for debugging purposes and represents the actual software breakpoint on x86 [59]. It generates only one byte code that consists of the opcode `0xCC` and is responsible for calling the debug exception handler. The reduced size of one byte is reasonable for inserting software breakpoints in the executable code. The `INT3` behaves as a normal function call, except that it is also pushing the current processor flags to the stack. The debug exception handler returns to the normal execution of the program by popping the flags again.

- **Hardware Breakpoint**
  Hardware breakpoints are implemented by dedicated registers that keep a breakpoint address or value. They are more powerful than software breakpoints as they

allow to trap the processor on, for example, reads, writes or executes of an address or simple value. Hardware breakpoints are fast but limited to the typically low amount of debugging registers on a processor. Using them is also highly dependent on the design of the underlying processor architecture.

In x86, hardware breakpoints are implemented by eight registers (debug register 0 to 7) which are responsible for interrupting the processor on particular conditions. They can be accessed and configured by dedicated move instructions that allow to read and write them. All debug registers are privileged resources and can only be accessed in certain processor modes [59].

- **Breakpoint Address Registers**
  Intel specifies four debug registers `DR0`, `DR1`, `DR2` and `DR3`. They are the core of the breakpoint, since they hold the linear address or value that causes the interrupt. Based on the number of those register, up to four breakpoints can be used by programmers at a time.

- **Debug Registers**
  The registers `DR4` and `DR5` are used when debug extensions are enabled. Otherwise they are aliases for the registers `DR6` and `DR7` and do not have any particular meaning.

- **Debug Status Register**
  `DR6` is the debug status register. On the interrupt handling routine of the debugger this register can be read to get a closer information about the triggered interrupt. It contains flags indicating that a breakpoint condition was detected as specified in `DR7` or that an access of a debug register, a single step event or a task switch occurred.

- **Debug Control Register**
  `DR7` is the debug control register. Together with the breakpoint address registers it forms the actual breakpoints. Programmers have to set it up correctly to enable the breakpoint address registers. It contains flags, for instance, those indicating the type of each breakpoint (trap on read, write or execute) and if a breakpoint is enabled.

2.2.1.3 Layout of the Stack and Procedure Calls

Reverse Engineering of software, or working directly above the hardware layer in general, requires thorough knowledge about the layout of the stack and how procedure calls are implemented in assembler on the given hardware. On x86 this topic is probably more complex or at least more confusing than on many other architectures, as there are many different calling conventions for which the usage is depending on the compiler.

The calling convention specifies how procedures are called, how arguments are passed, who is responsible for preserving registers (caller or callee), who cleans the stack and how subroutines can return their results. Intel itself does not dictate a specific standard, but they propose different possibilities of how to realize function calls on their architecture.

The fundamental requirement of implementing function calls is the presence of a stack. On x86 the stack is a contiguous array of memory locations. Items are placed on the stack by the instruction `PUSH` and removed by `POP`. The general purpose registers `ESP` holds the address of the top of the stack and is called the stack pointer [59]. Pushing items to the stack decreases the memory address of the stack pointer which means that the stack grows down in memory. In order to realize different levels of function calls, the stack is split into stack frames which contain for example local variables or function arguments. The general purpose register `EBP` is specified to hold the address of the beginning of the current frame. This way, variables on the stack can be easily accessed by heaving a static pointer. A new frame is generated by pushing `EBP` to the stack and moving the content of `ESP` to `EBP`, the reversal is achieved by popping moving frame to stack pointer and popping the previous frame pointer.

The actual function calls are then performed by the call instruction. Two different type of calls, a near and a far call, are provided [59]. The near call is used for routines in the same task or functions that are close to each other based on their memory address. Based on the variable length encoding of instructions this can either be an address from a memory location, a register or an immediate offset value. The call jumps to the target address and pushes the return address to the stack. Popping the address back to the instruction pointer with the `RET` instruction results into a return to the original code. The far call is responsible for calling kernel functions or routines from different tasks. It jumps to an address in a specific memory segment and pushes the return address and the return segment to the stack. On the return of the subroutine, the instruction pointer is set to the return address in the saved segment. The arguments for a subroutine call can be passed either on the stack or in general purpose registers except for `ESP` and `EBP`. The content of registers can be preserved by pushing them to the stack. The easiest and most memory consuming technique is to use the instructions `PUSHA` and `POPA` that push and pop all registers from the stack. The function prologue, the first few instructions of procedures that are always the same, is responsible to create a new stack frame and allocate space on the stack. Depending on the compiler, the type and size of instructions inside the prologue can vary, but usually are a push of the frame pointer, a move from the stack pointer to the frame pointer register and an instruction that allocates the space needed on the stack. This gives a size of six bytes per prologue.

2.2.2 The ARM Architecture

Besides x86, the ARM architecture is occupying large parts of the processor market with a significantly different design and hence, area of application. ARM is an abbreviation for *Advanced RISC Machine*. It is a *RISC* processor architecture that supports implementations across a wide range of performance points. The architecture comes with three different profiles (*A, R and M*) that focus on the domains performance, hard real-time and low power consumption [76, p. 30]. The recent architecture revision is *ARMv8* which is fully backwards compatible with older revisions. Currently, ARM is dominating the processor market of mobile computers like tablets or smart phones, but is also evolving in other areas with a total market share of 32 percent in 2015 [94, p. 16].

As a *RISC* architecture, ARM builds upon a simplified instruction set that merely uses atomic instructions with a regular encoding. Atomicity means that almost each instruction takes only one processor cycle. Regular encoding implies that all instructions can be described by a few simple patterns. ARM instructions have a fixed length of 32 bits [76, p. 78].

Depending on the execution mode of the processor, instruction can also have a variable length in Thumb mode for example it is of 16 bytes. Thumb is a processor execution mode that allows to optimize code density for microprocessors with few resources. For simplicity, the instruction uses a load and store model [76, p. 30]. This means that all operations are performed on processor registers and not directly on the main memory. Dedicated load and store instructions handle the data transfer from main memory to registers and the other way around. Choosing this design reduces the complexity of the instruction set and enables compilers to optimize register allocation. In the recent revision, the architecture is extended to support 64 bit by introducing two modes of execution. 64 bit mode is a superset of 32 bit mode that allows to access more registers, for example, registers with a larger size or additional instructions. Since our evaluation board only supports 32 bit execution mode, in the following sections, we will focus on the properties of the 32 bit architecture, not on 64 bit nor Thumb mode.

Besides the decoding unit for instructions, ARM also focused on inventing technologies that enhance the power-efficiency of processors under specific environments. In 2011 ARM presented the power optimization technology *big.LITTLE* that allows to build processors with cores of different sizes [77]. This way, depending on the load, the processor can switch between low power consumption and high performance to adapt to the needs of a longer battery runtime and still have enough computational overhead for CPU-intensive tasks, which is for example beneficial for smart phones.

2.2.2.1 Coprocessors and Application Level Registers

The whole design of the ARM architecture is based on different coprocessors. Coprocessors do not necessarily have to be separated in hardware, they are more like a logical subdivision of the processor. The coprocessor that represents the main module of the processor is responsible for the instruction fetch and implements the basic instruction set with all standard arithmetical and logical operations. Additionally, ARM defines up to 16 coprocessors that extend the functionality of this main module [76, p. 3863]. Coprocessor 0 to 7 are implementation defined, coprocessor 8 to 15 are reserved by ARM. This means that ARM only specified the interfacing for the first eight coprocessors and the implementer of the chip chooses the specific functionality. Contrarily, the ARM reserved coprocessors are fully specified in the instruction manual and provide, for example, an interface for debugging (CP 14) or advanced SIMD instructions (CP 10 and CP 11).

On application level, programmers can access different registers that are available on each coprocessor. This set of general purpose registers is also available in any processor mode [78, p. 14]. In 32 bit execution mode 16 registers each with a size of 32 bit are available, in 64 bit mode this set is extended to 32 registers with a size of 64 bit. Since we focus on the 32 bit version, we are interested in the following registers:

- **General Purpose Registers**
  Registers 0 to 12 are general purpose registers and can be used by programmers freely. Depending on the compiler and the implemented procedure call standard they may have additional properties, but from the architectural side of view they are equal.

- **Stack Pointer**
  Register 13 in principle is another general purpose register. Caution should be used for applications based on a stack, since ARM suggests to use this register for holding the address of the top of the stack. Otherwise this register can be accessed and used as any other general purpose register.

- **Link Register**
  Register 14 is a similar special case as the stack pointer register. It is also a general purpose register, but is referred to as link-register. On execution this register is dedicated to hold the return address of the current subroutine call.

- **Program Counter**
  Register 15 is a specific register that contains the program counter. Even though the instruction set provides general access to this register, ARM deprecates the use for any purpose other than the program counter. This register reads the address of the current instruction plus 8 which means two instructions.

2.2.2.2 Breakpoints on ARM

The concept of debugging on ARM is in general similar to that of other architectures, only the detailed implementation varies. ARM also implements software and hardware breakpoints which were described in section 2.2.1.2. Debugging is under the responsibility of coprocessor 14 [76].

- **Software Breakpoint**
  ARM implements software breakpoints as *Breakpoint Instruction Exceptions*. A dedicated breakpoint instruction `BKPT` throws this exception on execution, independent on the current execution mode, processor privilege level or exception level. This means breakpoints instruction exceptions cannot be turned off and always raise on execution. The insertion and deletion of breakpoints from executable code is simple as all instructions have a fixed length of four bytes. The insertion then boils down to a substitution of instructions.

- **Hardware Breakpoint**
  The ARM specification defines up to 16 hardware breakpoints per processor. The actual number of available breakpoints is chosen by the implementer and is accessible from the *Debug ID Register* (`DBGDIDR`). A triggering hardware breakpoint throws a *Breakpoint Exception* that interrupts the execution of the current program and steps into the exception handler. Hardware breakpoints can be turned off, trigger on access of data or simply at execution of an instruction. This is realized by assigning each hardware breakpoint a pair of registers. A control and a value registers:

  – **Breakpoint Control Register**
    The registers `DBGBCR0` to `DBGBCR15` hold the control information of each breakpoint. They hold flags that disable or enable the breakpoint, define the triggering type like for example on address match, address mismatch, context id match or mismatch, and for which exception level the debug event is generated. Depending on the implementation, a breakpoint might only support address matching. Each breakpoint additionally can be linked to another breakpoint. That means that only if the conditions of both breakpoints are met, the debug exception is thrown. These registers can be, similarly to all other registers of the debug coprocessor, be accessed by the coprocessor interfacing move instructions `MRC` and `MCR`.

  – **Breakpoint Value Register**
    The registers `DBGBVR0` to `DBGBVR15` hold the value associated with the breakpoint. Depending on the selected conditions in the control register, this can be either a virtual address of an instruction or a context id. This register together with the corresponding control register forms the actual breakpoint.

2.2.2.3 Layout of the Stack and the ARM Procedure Call Standard

The concepts of the stack on ARM are similar to the stack described in section 2.2.1.3. It is also a contiguous array of memory locations that is fully descending. That means, the address of the top of a growing stack decreases in address space. In 32 bit mode, the stack must be aligned to four bytes. The stack is accessed with the `PUSH` and `POP` instructions that are aliases of `STMDB` and `LDM`. These instructions allow to push and pop multiple registers from the stack at a time. As a result, only one instruction is required to save and load multiple registers. Both instructions also allow to directly access to the program counter, which for the case of a load results into a branch after execution. The address of the top of the stack is stored in register `r13` which is the `Stack Pointer Register`. For programming languages that use stack frames there is no dedicated register that holds the frame pointers. This means there is a subdivision of the stack in different stack frames, but the program needs to manage these frames on its own.

For function calls, ARM recommends to use its specified procedure calling standard. Since direct modification of the program counter is deprecated, the only real choice to call functions is to use the branch and link instruction which allows to move to addresses in a 32 MB memory block relative to the current address. As the name already implies, branch and link does also store the program counter before the branch is executed into the `Link Register`. This makes it possible to return to the calling routine by simply reading the address from the `Link Register` into the `Program Counter`. The ARM procedure call standard defines that the arguments on a procedure call are passed in the first four registers R0 to R3 [78, p. 17]. If the number of arguments is greater than four, the remaining arguments are pushed to the stack. Also, if the size of the arguments is larger than 32 bit, the arguments are passed by a pointer and stored on the stack. These four registers are also used for returning the values of the subroutine. When the return value is 32 bit or less, only R0 is used. Subroutines must preserve the registers R4 to R8 and R9 to R11, since they are defined to hold local variables.

2.3 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) is a method where instrumentation code is injected into a binary application to analyse its behaviour during runtime [83]. Once injected, the instrumentation code will execute like it is part of the normal execution stream inside the application. The instrumented code is generally transparent to the application to which it is injected. Being able to analyse an application during runtime means that it is possible to understand the behaviour and state of the application at various stages it goes through during actual execution. This means there is a possibility

to get an idea about the functions and modules the application calls within its own program as well as some others it references from the operating system itself. Hence dynamic binary instrumentation operates on what actually occurs instead of assuming what might occur like static binary analysis does. Even though DBI is not exhaustive, it does provide a detailed analysis of what an application goes through during its actual execution state. Whether a DBI based program analysis tool is usable or not is generally determined by the speed of the tool. Most developers have an interest in performance improvements of the instrumentation [42].

Some advanced tools which perform tasks like detecting memory allocation errors [82], model system performance [89], cache simulation [61] and detecting violations in the security of a system [68] have been built using DBI techniques.

There are different approaches to do DBI. One way is the probe-based like Dynist [21]. In this approach so called trampolines are added in the executable and when they are executed, they jump to the instrumentation instructions. This makes the code not transparent because the original instructions are overwritten. A program can notice this and hence prohibit reverse engineering. But as an advantage, trampolines can be executed fast without much overhead.

The more flexible approach for analysis is the just-in-time (JIT) based approach. This means that the framework does the main work during the runtime of the analysed application.

2.3.1 Pin

An example for an instrumentation framework is Intel's Pin. Since Pin 2 was released in July 2004 for multiple architectures (x86, x64, Itanium and ARM), it became a popular tool for analysing applications at run time. This section is based on the paper *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation* [79].

Using Pin requires not only the Pin instrumentation system, but also a Pintool, written by the user. Pintools basically consist of 2 types of routines. *Analysis routines*, which perform the desired analysis work and *instrumentation routines*, that determine when the analysis routines are called.

A simple example would be an analysis routine with a counter and an instrumentation routine which places the analysis routine at every procedure call of the instrumented application.

To allow different analysis methods, Pin supports different levels of instrumentation granularity. The instrumentation routines can be called for each *instruction, trace, or procedure* and additionally for certain events concerning the *image*, like loading or unloading additional images.

2.3.1.1 How Pin works

Pin can instrument an application both from launch time and during execution. In both cases, Ptrace (for Unix) loads Pin (similar to a debugger), which in turn loads a Pintool into the application's address space. From there, the JIT (just-in-time) compiler intercepts the currently executing code and generates new code with added instrumentation instructions.

The purpose of the JIT compiler is to not recompile the complete application, but rather to split the application up into smaller parts, called *traces* and save them in the *Code Cache*, as seen in figure 2.1. These traces are split up either at a certain *number of instructions*, at an *unconditional branch* (call, return) or after a number of *conditional branches*. A trace can have multiple exits, created by *conditional branches* inside the trace, who either point to another trace or a *stub* (basically a trace, which has not yet been compiled). These stubs remain stubs until they are actually executed, at which point the JIT replaces it by a trace. So, running the application together with Pin attached, it is compiled on-demand, trace-by-trace, where a trace is only compiled once & the original code of the application is not executed anymore.

Pin's architecture is basically a virtual machine, whose purpose consists in using the JIT compiler to generate code, caching it in a code cache, and providing an API for developers to write Pintools and instrument applications. This can also be seen in in figure 2.1.

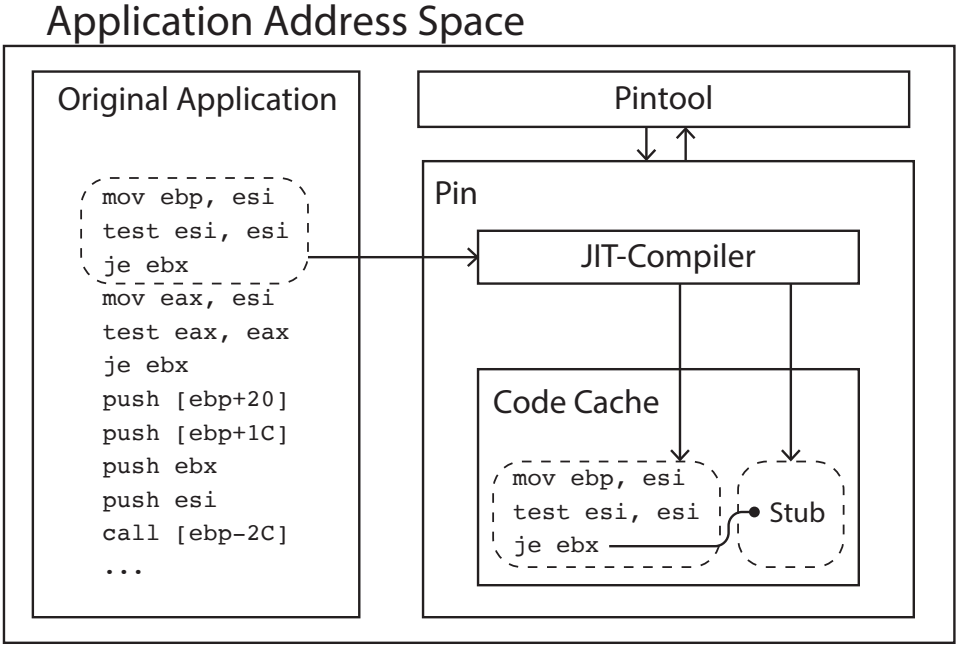


Figure 2.1: Pin's architecture, based on [79]

For improved performance, Pin also offers probe mode [58]. It differs from the regular (*JIT*) mode by not using the VM, but rather injecting code in the original application, which results in a speedup. A drawback of this method however is that only procedure granularity is available.

2.3.1.2 Pin's optimizations

To improve the JIT compiling, Pin uses multiple optimizations: *Trace Linking, Register Re-allocation and Thread-local Register Spilling*, among others.

Linking Pin's *traces* usually end in a control transfer, which could involve the VM generating the next trace from a stub. *Direct* control transfers (with only 1 jump target) can be easily improved by simply placing a jump at the end of the trace. *Indirect* control transfers (Jump, call or return) however require more effort because of multiple jump targets for the trace exits. Pin's solution involves predicting multiple jump target address and building a chain of comparisons which try to find the correct jump address when that part of the code is reached during execution.

Register Re-allocation Processes like the creation of the chain in the *Trace Linking* involve utilizing registers, which are usually already in use by the application. Thus, we need a way to free registers to allow these processes. A live analysis for the registers is possible, which Pin does incrementally, since a complete flow graph is not available when using JIT compilation.
Pin also has to make sure that the register bindings at the trace exits are what the linked trace expects at its beginning. The simple method is to simply dump all register contents to memory and restore them to the correct registers for the next trace. However, this is too inefficient for Pin. Instead, when Pin compiles a trace's code for the first time, it matches the register layout to the one from the previous trace, so no flushing to memory is needed. But in future executions of the trace, the expected registers of the trace may change, which Pin solves by creating *reconciliation* code for the affected registers. This reconciliation code is responsible for reordering the registers to match them to the expected order for the next execution.

Thread-local Register Spilling The need of extra memory for the registers is not completely eliminated, for example the reconciliation might need memory for the exchange of registers. This is solved by reserving one register (%*rbx* on x86) as the *spill pointer*, pointing to the memory location we use for our purposes. Since threads can be switched at any time, each thread has its own spill area and the value of the spill pointer is adjusted for the next thread's area.

2.3.1.3 Instrumenting Windows applications

Similar to its Unix version, Pin also injects its Dynamic Link Library (DLL) and the user's Pintool DLL into the target process. Launching an application with Pin attached involves the following steps [103]:

- Creating the process and attaching to it until kernel32.dll is finished initializing
- Immediately afterwards, changing the instruction pointer to a small routine to load Pin's and the Pintool's DLL.

Injecting so early maximizes the information gain of the instrumenting process.

System calls like thread creation/termination also have to be handled by Pin. It intercepts the system calls, and executes them in the Pin environment. Special care must be taken to recreate the original environment of the system call regarding registers, arguments and the system call number. As a special Windows case, the system call numbers are undocumented and can even change in between Windows versions. So, Pin recreates the numbers dynamically at runtime using a dummy process and executes each system call once and records the results [103].

Handling the Windows Exceptions works similarly, Pin has to match the exception's data structures to the instructions in the code cache. In particular, the exception context has to be matched to the original registers, since Pin may have altered registers during execution of the program, as described in section 2.3.1.2. This retranslation has a large overhead (200000 cycles instead of 5000) and can lead to a performance loss in applications which use exceptions heavily [103].

Pin monitors the threads of the application closely and intercepts their execution as soon as they are created. Additionally, Pin keeps a shadow copy of the stack, since applications could theoretically operate beyond the top of the stack. Pin also creates a VMM thread object for each thread, which are used for thread-local data for Pin (register spill area, active system calls) and for the Pintools [103].

2.3.1.4 Controlling applications with Pin

It is possible to use Pin to execute the instrumented application at any point in the application by using the *CONTEXT* and *CHECKPOINT* APIs [88]. The checkpoint system allows execution at a specific point (which was marked before with a checkpoint) with the current application state, including the currently set register values. The context system is more powerful and allows editing the registers before the execution at another point. The context system's drawback is the performance cost, with the context switch taking 4 times longer than with the checkpoint system. Potential uses for these APIs are a multi-threading library and a transactional memory system [88].

2.3.1.5 Pin's data buffering

Pin's instrumentation unfortunately also includes a performance overhead, especially when a large amount of data is collected by the instrumentation. One improvement for this problem is provided by Pin's buffering system [114]. The buffering system provides a new API for Pintools, optimized buffers and a fast method to detect overflows. The Pin developers claim a 4x improvement over existing buffering methods with the old API [114].

2.3.2 DynamoRIO

DynamoRIO [18] is the framework with the longest development history of the three discussed frameworks. It initially originated from Dynamo which was developed in the nineties for a RISC architecture. In 2000, with the help of the Runtime Introspection and Optimization (RIO) group, the code was ported to the x86 platform which resulted in the new name, DynamoRIO. VMWare acquired it in 2007. Currently Google actively contributes to DynamoRIO [34]. The framework supports x86 (32-bit and 64-bit) and is available for Windows and Linux. It is developed as a free software under the BSD license.

DynamoRIO is an interpreter for IA32 instruction set. It has an IA32 platform specific implementation. At an individual instruction set granularity, DynamoRIO allows modification and optimization as well as dynamic introspection of native executables. DynamoRIO is capable of running on binary files which have not been modified and need no special support from any hardware or the operating system.

Since binary instrumentation of the original source code has a huge overhead DynamoRIO uses a cache where it copies basic IA32 instruction blocks from the instrumented program to interpret and execute them there. This process is called copy-and-annotate. The control flow instruction at the end of each block is modified so that the control is returned to DynamoRIO which can then find the next block which needs to be executed. Code is only allowed to execute inside the cache controlled by DynamoRIO and not within the original binary image. To save overhead, DynamoRIO makes sure that blocks with direct branches are sent straight to the next basic block instead of passing control back to DynamoRIO. To deal with indirect jumps, DynamoRIO creates a hash table lookup in the cache which converts the address of the indirect jump into the corresponding address in the cache. The control is returned to DynamoRIO only if there are no cached addresses for the indirect call in the hash table.

The traces generated by DynamoRIO enter only from the top but may end up exiting from many places. DynamoRIO also has an interface which gives external libraries access to modifying traces and basic blocks prior to writing them in its own cache. This

allows instrumentation and optimization of the program and DynamoRIO allows the application to run under itself through an interface [33].

DynamoRIO puts a lot of effort on transparency but it also has limits. For example, there might be a scenario where the native execution generates an exception because it reads from an unmapped memory space. However, inside the virtualization environment of DynamoRIO it might happen so that the same memory read is at a location where DynamoRIO stores data which needs to be accessed by the code cache execution. In this case, no exception will be generated. Also memory intensive 32-bit applications can crash because DynamoRIO has a memory overhead which can easily exceed the addressable memory [20].

2.3.3 Valgrind

Another framework to create tools for dynamic binary instrumentation programming is Valgrind [83]. Valgrind is an Open Source Framework under the GPL license and is available for many different ISAs like x86, ARM, PPC and MIPS. To combine the different ISAs Valgrind uses an intermediate representation. Valgrind takes up the responsibility of instrumentation and inserts code at runtime into existing machine code. Valgrind has a core section to which tools written in C are written as plugins. So to explain it really simply, the Valgrind core along with the plugin tool creates the Valgrind tool.

This Valgrind tool is responsible for loading the application being instrumented and inserts itself into the process as the client starts up. Valgrind works in a just-in-time compilation mode. After the client is loaded, Valgrind executes the client code block by block by compiling the code once more. During this compilation process, the machine code from the client is disassembled into another intermediate code representation, namely, UCode. The plugin now instruments the generated UCode and the resulting instrumented code is once again converted into x86 code and stored in the code cache of Valgrind. This can now be rerun as and when it is necessary. Most of the time spent by the Valgrind core is to find and run translations. The client's original code is never run. Other services like error recording and logging of debug information is provided by this code as well. Valgrind allows the execution of a single tool at a time.

Also with Valgrind, it is possible to create shadow value tools and shadow every memory or register, which means, a value can be assigned to every memory address or register. Valgrind fulfils all the requirements for building such tools. First, it must be possible to have a shadow state of the registers and of the memory which contains every value of the original state. Secondly it must be possible to instrument all the reads and writes of the system calls. Allocation and de-allocation must also be handled like it is done using mmap. Finally, to keep the execution transparent, it is important that shadow values are

provided but it should also be possible to output some extra values. These requirements are partly supported by DynamoRIO and Pin through virtual registers.

Valgrind takes many basic blocks and combines them together into a superblock. To reduce the management overheads, these superblocks have only one entry but can have multiple exits. Valgrind is one of the most flexible DBI frameworks which also means that is has a few drawbacks. [84]

2.4 Classical Format String Attack

To give the reader a background in the topic of this work, we begin by describing the classical Format String Attack (FSA) attack that has evolved throughout the recent years. The classical FSA exploits the behaviour of `*printf` functions, which correspond to a class of functions that use formatting information to specify the output format. Since the `printf` function family are variadic c-functions, the number of format specifiers within the format strings is controlling the number of parameter which are used by the function and are thus popped from the stack[1]. The most important format specifiers for exploiting format string vulnerabilities are listed below:

%x - pop address from stack

%s - pop address and dereference

%n - write printed char count to address on stack

%hn - write to lower 16 bits (short)

%hhn - write to lower 8 bits (byte)

A basic format string vulnerability just passes a single argument to the `printf` function. In the classical exploit the buffer is defined as a character array on the stack. If the buffer contains user controlled input, an attacker can fill this buffer with arbitrary format specifiers, as listed above, and the function will access the next immediate value on the stack for each format identifier within the buffer.

Depending on the used specifier, different actions will be executed on the stack. The attacker can, for example, shift the stack by using a *%x* operator or dereference a memory address to access the content that is referenced by that address by using the *%s* operator. But the most important format specifier for having a generic way for exploiting this vulnerability is the *%n* operator. This specifier takes an address from the top of the stack, dereferences it, and writes the total number of printed characters into the specified location. This allows an attacker to write arbitrary values to an arbitrary

[1] e.g. *printf("id: %d, size:%d, name: %s",id,size,name)* consumes three arguments.

memory location, assuming that the vulnerable input buffer is located on the stack[2]. The chosen address could be the address of the saved return value, the pointer to an address in the Global Offset Table (GOT) or an entry in the list of the destructors (*.dtors*). Thereby the attacker is able to change the control flow of the application, if he redirects such a pointer to some shellcode that has been prepared in advance.

To protect against this type of attack, protection mechanisms have been established to mitigate memory write attacks. We hereby differ between two classes of protections mechanisms: Compiler-based and system-based protections. A commonly used compiler-based defence mechanism against control flow violations are stack cookies. The basic idea behind stack cookies is that in order to overwrite the return address of a function, a user has to overflow a buffer on the stack and thus overwrite everything between this buffer and the return address. If the compiler places a stack cookie between the buffer and the return address, the attacker also has to overwrite this cookie. As an attacker is unable to know the content of this cookie in advance, it is possible to detect the modification of the return address, if the cookie was overwritten by an attacker. This cookie can be easily circumvented by FSAs, because the place to be written can be directly controlled by the attacker. Another compiler-based protection mechanism is the compiler flag RELocation Read-Only (RELRO). This mechanism is resolves all addresses at the beginning and maps the GOT as read-only, so an attacker cannot overwrite the function pointer and redirect the control flow. A further mechanism that specifically protects against a FSA is using the `FORTIFY_SOURCE` option at compile time. The idea behind `FORTIFY_SOURCE` is to check the source code for the usage of certain insecure functions. These are common functions (e.g. `strcpy`) that use a given buffer and expect it to be delimited by a null terminator, which is not always the case. If the compiler detects the usage of such an insecure function (like `strcpy`), it tries to identify the size of the destination buffer and replaces the vulnerable function with a more secure function. A call to the `printf` function is replaced with a more secure function, so that the compiled program can handle a possible attack at runtime. If an attacker, for example, tries to use the `%n` parameter in a format string, the program will crash.

Although this is a good idea, Planet has shown that this protection can be circumvented by overwriting the *IO_FLAGS2_FORTIFY* bit in the file stream by controlling the `nargs` value in the format string [93]. Another compiler-based protection is pointer encryption. This technique is used to encrypt instruction pointers with a simple encryption function, which is not known by the attacker and thus prevents a pointer manipulation by the attacker [29]. This approach is thereby somehow similar to the stack-cookie approach. Although even if the algorithm uses XOR, the attacker can easily find the key if he

[2] An input to a buffer like *"\x78\x4f\x9e\xbf", "%5u", "%10$hhn"* will, for example, write the value $0x9$ to the least significant byte at the address $0xbf9e4f78$, because in this example the tenth value on the stack is containing our user input.

knows a pair of plain and cipher text. Furthermore, instruction set randomization uses the same idea in which the attacker does not know the instruction set [41].

As next, we describe common system-based mitigation approaches. One approach is Address Space Layout Randomization (ASLR). ASLR randomizes the memory addresses of both the executed code as well as the stack. Unfortunately, it only randomizes the prefix of entire pages, thus in case of $4K$ pages (which is common on the Intel architecture), the last 12 bits of an address are not randomized. On modern systems we also see more randomization added to some mappings. They extend it to 20 bits and therefore only the last 4 bits are not randomized. This does not ensure security in 32 bit systems because the address can still be bruteforced. The reason for this is the limited number of randomized bits [11, 12, 81, 102, 108]. Another system-based protection mechanism is Non Executable Bit (NX) or Data Execution Prevention (DEP). Its goal is to hinder the execution of code that is located on a page that is supposed to contain data. Thus it hinders an attacker to prepare, for example his shellcode on the stack or heap.

Libsafe is a library, which can be used to protect against overwriting the stack at run-time. Equal to `FORTIFY_SOURCE`, it replaces vulnerable functions like `*printf()` with secure versions. If there is a possible attack the library will kill the process and log the event. The disadvantage of this approach is that it works only for limited amount of functions [9].

FormatGuard is a patch for glibc, which counts the arguments that are given to the printf function at run-time and compares it to the number of format specifiers (%). If the format string uses for example more arguments than the actual number of printf arguments, then it is assumed that an attack has taken place and the program will be terminated. To use this protection the programmer has to re-compile the program with FormatGuard. A problem with this approach is that it can detect the attack only if the number of specifiers is changing but not if the variables are reordered. Indeed, this kind of attack cannot be recognized by FormatGuard [28].

Chapter 3

Related Work

In this chapter, we discuss the related work on security testing and intrusion detection in encrypted environment. This work is related to the Chapters 5, 6 and 7.

The related work on format string attacks is described in Chapter 4 and we provide the related for function identification in Chapter 8.

3.1 Security Testing

Many existing fuzzing frameworks facilitate the security testing of network communicating applications. Gascon et al. [45] present a fuzzing framework for proprietary network protocols which uses inference to create a generative model for message formats. Their approach relies on unencrypted network traffic, similar to many other smart automated model-based [69] [8] [49] and grammar-based [119] [46] fuzzing techniques. Nowadays, there is also a vast amount of powerful commercial fuzzing and vulnerability scanning frameworks like Defensics [24], Nessus [100], beSTORM [99], Peach Fuzzer [36], honggfuzz [106] and american fuzzy lop [120] available in the market. They provide very complex and sophisticated algorithms to cover many different areas of fuzzing and vulnerability testing, but overall also lack proper support of encrypted network communications.

Biyani et al. [13] address this issue and present a solution by extending the SPIKE fuzzing framework to support encrypted protocols. They add a SSL wrapper to the existing plaintext fuzzer which allows to communicate with the target test application over an encrypted tunnel. This way, the fuzzer can inject its plaintext test data into the encrypted channel and test the target application for vulnerabilities. This approach, however, is limited to SSL encryptions which only represent a small part of proprietary

software products. Another drawback is that their implementation is customized and only applicable for the open source fuzzer SPIKE. Tsankov et al. [113] introduce a different solution that allows a more generic fuzzing of encrypted protocols. Their approach is based on the knowledge of the encryption key and algorithm, which is problematic from a security point of view

As of yet, there is no generic and security preserving solution to testing of applications with encrypted network traffic. We propose an interface for testing frameworks. It makes the encryption of the program under test transparent without violating the security of end-to-end encrypted communications. This way, we reduce the problem of testing encrypted protocols to testing of plaintext protocols and thus, enable the usage of many already existing testing tools.

3.2 Intrusion Detection

There exists a variety of Intrusion Detection Systems(IDSs) in order to detect attacks over network communication. Li et al. [75] introduced different concepts and common standards regarding IDS, providing a theoretical base for further research. They introduce a general categorization into signature-based (SID) and anomaly-based (AID) intrusion detection. While SID may detect known intrusions reliably, novel intrusions remain unobserved. AID may also detect new intrusions but, amongst others, suffers from high false alarm rates. They also state that network-based IDS do not detect anomalies in encrypted network traffic. With a growing percentage of encrypted network communication, the research of IDS in encrypted networks becomes more prevalent.

3.2.1 Intrusion Detection in Encrypted Networks

Intrusion detection in encrypted networks presents a challenge, which is tried to cope with in a variety of ways. Existing research on this topic can be classified into two main categories. A considerable amount of strategies concentrates on traffic analysis of the encrypted data, without decryption or plaintext inspection. Fewer approaches employ decryption or extraction of decrypted data for analysis. While traffic analysis approaches are limited to interpretation of transmission information such as packet size, timing or the encrypted data itself, these strategies do not need to solve the problem of extracting plaintext from the observed communication and respective emerging problems. One of these problems is keeping the end-to-end encryption scheme intact and securing the possible attack surface of decrypted information being present in the IDS. Strategies, employing decryption or plaintext extraction, however, enable a more precise analysis of the communication, as all communicated content is available for dissection.

An example for basic traffic analysis IDS is the system proposed by Joglekar and Tate [62], which only focuses on protocol violations for intrusion detection. The percentage of detectable intrusions is therefore reduced to this specific kind of attack. Yamada et al. [118] and Foroushani et al.[40] present traffic analysis approaches based on data size and timing. While the first detection approach is limited to attacks which employ a scanning phase at the beginning, the second approach focuses on SSH connections to public servers only. Koch and Rodosek [70] [71] explored analysis approaches such as command evaluation and user identification in encrypted remote sessions (e.g. SSH/SSL) through packet sizes, divergences and communication delays. While the first approach, command evaluation, limits intrusion detection to encrypted remote sessions (e.g. SSH), the system proposed in the second approach (S2E2) was not tested. Another behaviour-based approach was put forward by Koch et al. which employs detection through similarity measures, by means of correlation to the majority of connections. Augustin and Balaz [7] introduced a hybrid detection approach combining recognition of applications used in SSL encrypted communication and anomaly-based detection. This limits the results to attacks through SSL encrypted communication. Flow-based detection systems have been introduced by Hellemons et al. [55], focusing on SSH traffic only, and Amoli and Hämäläinen [4], proposing a flow-based approach for high-speed networks using machine learning, but providing no test results. Zolotukhin et al. [121] proposed a data-mining approach employing the DBSCAN algorithm, limited to DoS attacks. Another technique, applicable for encrypted traffic analysis in an intrusion detection scenario, is proposed by Böttinger et al. [16]. They demonstrated the feasibility of detecting fingerprinted data in encrypted TLS traffic for large amounts of traffic, restricting detection of small payloads and other forms of communication. Generally, the results of many traffic analysis based approaches are limited to a specific form of encrypted communication, e.g. SSL, or attacks.

Strategies, employing decryption or extraction of decrypted data, are put forward by Abimbola et al. [3], which employs extracting plaintext data from the operating systems protocol stack and Goh et al. [48] [47], mirroring encrypted network traffic to a central entity, which decrypts and analyses the data. However, this dedicated decryption of data in a third entity breaks the end-to-end encryption scheme. Many frameworks are designed as a Host based Intrusion Detection System (HIDS), where the system is executed on the same host as the observation target. This is a challenge regarding protection from attacks against the framework. Other systems [3] are designed to extract information from lower layers of the network protocol stack (e.g. OSI layer 3), requiring more incisive modifications to the observed host.

Our approach provides a general detection strategy, independent of encryption protocol or attack type. We extract the monitoring data directly from the applications process address space via Virtual Machine Introspection (VMI), minimizing modifications to the target host. This improves the security of the framework compared to conventional approaches. As our system serves as an observation interface for an external IDS by

making the encryption transparent, conventional IDS solutions for plaintext traffic can be employed.

3.2.2 Virtual Machine Introspection for IDS Protection

With virtualization providing a variety of separative and protective means, the usage of VMI for securing IDS has been proposed in several publications. Garfinkel et al. [43] move the whole IDS to the Virtual Machine Monitor (VMM), using VMI for monitoring. Deng et al. introduce stealthy binary program instrumentation via hardware virtualization in their project SPIDER [31]. The approach was implemented for KVM, using software breakpoints to trap into the hypervisor. Originally designed for malware analysis, Vasudevan and Yerraballi introduce their implementation of VAMPiRE [115], using stealth breakpoints through virtual memory and hardware single-stepping techniques. These solutions introduce some drawbacks. The usage of software breakpoints for application inspection is undesired under some circumstance (e.g. for malware analysis [115]), while single-stepping through hardware breakpoints introduce a significant performance overhead. Ho et al. [57] describe their implementation of a pervasive debugger implemented for the hypervisor Xen. The PDB is located on the hypervisor level. The implementation of the pervasive debugger depends on GNU-Debugger (GDB) to set and pass breakpoint events to the PDB Client located on the lower level.

The approach introduced in this thesis is different. We use hardware breakpoints for target inspection but don't require single-stepping, thus avoiding performance cutbacks. Our solution combines the benefits of a general detection strategy and virtualization, providing an observation interface for multiple architectures with improved protection. We reduce intrusion detection on encrypted channels to intrusion detection on plaintext channels, thus enabling the use of existing IDS tools in the area of encrypted communication.

Chapter 4

Blind Format String Attacks

Intruders are exploiting software vulnerabilities to penetrate the system. If encryption data transfer is used, the confidentiality of the payload is protected, but an attacker can still act the same way as no encryption is in place. Unfortunately an IDS has difficulties to inspect the payload, and therefore can't detect this attack. In that case, only host base protection mechanisms can help to stop the attack. In this chapter, we select Format String Vulnerabilities(FSVs) as an example to investigate the effectiveness of the software mitigation.

FSVs are known for many years and are assumed to be easy to detect. But unfortunately there still exist applications, which are vulnerable to this kind of attack. According to the CVE database [110], the number of vulnerabilities that can be classified as a format string vulnerability has decreased in the last 15 years. Over the course of the last 8 years, however, it appears to stay on a constant level.

There was, for example, a severe format string flaw in the application *sudo* from versions 1.8.0 through 1.8.3p1, which was found in the `sudo_debug()` function (CVE-2012-0809). In Linux kernel through 3.9.4 existed a bug which allowed an attacker to gain privilege rights, which could be exploited by using format strings in device names (CVE-2013-2851). There also existed an exploitable format string bug in the Linux kernel before 3.8.4 in the function `ext3_msg()` which could be used to get higher privileges or crash the system (CVE-2013-1848). This vulnerability is even found in vehicles, e.g. the bluetooth stack of the car BMW 330i lead to a remote crash of the multimedia software (CVE-2017-9212). Therefore, we can assume, that format string bugs will still be present in the future. Table 4.1 lists the number of registered format strings vulnerabilities over the last 8 years.

Since the first methods for a FSA were released, system wide protection mechanisms like NX and ASLR are implemented in many operating systems. Also compiler-based

Year	2010	2011	2012	2013	2014	2015	2016	2017 (until may)
Number	14	9	18	14	5	8	8	5

Table 4.1: Number of format string attacks in the last eight years

protections like stack-cookies and FORTIFY_SOURCE should protect from binary exploitation. All these protection mechanisms make exploitation more difficult nowadays. Nevertheless, Planet [93] has shown that FORTIFY_SOURCE can be circumvented and Payer et al. [90] have shown, that NX can also be bypassed.

All generic exploiting techniques shown in the past are relying on two mature constraints. First, the input buffer that is used by the attacker has to be placed on the stack, and secondly, the attacker requires knowledge about the output of the format string. In this chapter, we instead assume, that the attacker is blind regarding to the output of the application. He will not receive any memory leakage by the exploited application. In addition, we also show that with our technique, the attacker's payload may also be located on heap, instead of the stack.

Parts of this chapter were published [66].

4.1 Related Work

The topic of FSAs is already known in the academic world for over a decade. The basic concept was first introduced by Newsham back in the year 2000 [85]. The concept was then extended and described in more detail in 2001 by Teso [98]. The attack has been enhanced by Haas [52] and Planet [93] in 2010. Haas is showing that the memory leak of a format string can be used to calculate all relevant memory address to build the exploit string without any bruteforce, whereas Planet is showing a way to bypass the `FORTIFY_SOURCE` protection using format string attacks. In the recent years, however, this topic gained less interest. Payer et al. [90], describes a method for applying both Return Oriented Programming (ROP) [101] and Jump Oriented Programming (JOP) [14] to format string attacks described by Haas and Planet and also discusses different protection mechanisms.

Since the current state of exploiting FSAs is based on a memory leakage, we will focus on the challenge of exploiting without memory leaks. Since our thesis is focusing on encrypted data communication, we consider the case of remote exploiting.

4.2 New Attack Methods

After we discussed the classical FSA in the last section, we now describe a novel technique to apply an FSA even in an environment, where the exploit string is placed on the heap and in addition the user has no direct control over the stack content. Afterwards, we will also describe, how it is possible to exploit this blindly, even without any feedback by the vulnerable program.

As this chapter is about describing a blind FSA, it is important to define the term "Blind Attack": A blind attack is a network-based attack that is executed remotely without any local access to the attacked system. In addition, the attack does not require the attacking entity to receive any data from the attacked system. In the case of a FSA this especially means that the output of the attacked `printf` function is not available to the attacker. Nevertheless, we assume, that the attacker is in possession of the executed binary beforehand. This is a legit restriction because most software is custom of the shelf software that is not self-developed and is available for the public.


```
1  void logfunc(char *buf) {
2      char * pch;
3      pch = strtok (buf,"|");
4      while (pch != NULL) {
5          printf(pch);
6          pch = strtok (NULL, "|");
7      }
8  }
9  int parse(char *buf, int log) {
10     if (log == ENABLELOGGING)
11         logfunc(buf);
12     /* Do something using local stack variables */
13 }
14 int handle(clientsocket) {
15     char *buf = (char*)malloc(SIZE);
16     //...
17     recv(clientsocket, buf, SIZE-1, 0);
18     parse(buf,1);
19     free(buf);
20     //...
21 }
22 int func(serversocket) {
23     //...
24     while(1) {
25         pid = fork();
26         if(pid == 0) { /* ... */ handle(clientsock); /* ... */
27         }
28         //...
29     }
30     //...
31 }
```

Figure 4.1: Format string vulnerability on the heap

4.2.1 Attack payload on heap

To exploit a FSA vulnerability, an attacker traditionally needs to store her attack payload in a buffer inside the vulnerable program. In Section 2.4, we have shown how a FSA is applied if the user input is saved on the stack. Within related work it is assumed that this buffer is located on the stack of the attacked system. This is an optimistic assumption, as it is not always the case in practice. The problem with a heap based FSA is that the attacker can only write to addresses, which are already saved on the stack using the *%n* specifier. In this case, the attacker is not able to write directly on the stack. This means, that the required destination address for FSAs cannot be placed on the stack, which stops the attacker to dereference this destination address using format string specifier.

Stack-based FSAs, however, rely on user controlled input on the stack. The attacker places the exploit string, which contains the address of the write destination, directly inside the user input buffer. This address can then be directly accessed by the *$* operator or using the *%x* operator many times to pop all values from top of stack until the attacker controlled data is at the top of stack. In our case we do not require the attacker controlled input on the stack. This means only application controlled data is referenced on the stack. We therefore assume that there is no other input channel to place data on the stack, which would make the exploitation easier.

4.2.2 Arbitrary write

Above, we described how to write to application controlled locations by dereferencing the memory addresses on the stack and writing to it using the *%n* specifier. Now we focus on a generic exploitation concept to achieve arbitrary writes into application memory. The basic idea of our novel approach is to use the saved frame pointer, which is stored on the stack once a new function is called. If the application is not compiled with specific flags like *-fomit-frame-pointer* every function will save/push the last frame pointer on the stack in the prologue and restore it in the epilogue. We benefit from this fact because this address is always pointing to another location on the stack, which is also writeable. Therefore, no protection mechanisms like NX, stack cookies or ASLR will protect the system from an attacker writing to that location. Whenever an application is using the saved frame pointer feature, one frame pointer is pointing to the next frame pointer like a linked list. The next frame is therefore also located on the stack on higher addresses, which can also be written to.

The goal of our mechanism is to use this list of saved frame pointers to achieve an arbitrary write to an arbitrary location within the system. With current FSA mechanisms we are only able to write to locations, which are already referenced on the stack of the current application. But by leveraging the linked list property of the saved frame

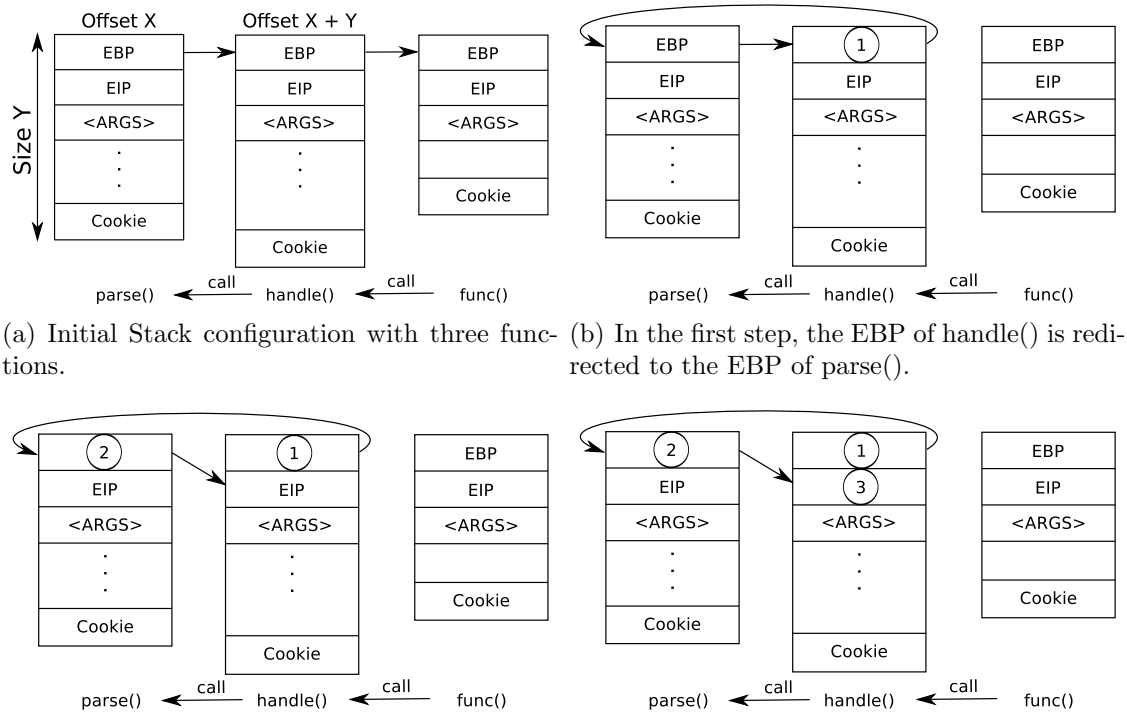
pointers, we are able to modify the saved frame pointers on the stack according to our needs and thus achieve a situation in which we are able to write to an arbitrary location in memory. First, we are using the saved base pointer (EBP) on a lower address to overwrite the value of the next saved EBP, to point it to an arbitrary address like the GOT. In the second step, we can write at this location with an arbitrary value.

4.2.3 Changing the control flow

As we are able now to write to arbitrary memory locations, we describe how it is possible to hijack the applications control flow using an FSA. This still requires exact knowledge about the addresses, which have to be modified in order to control the execution flow. In the case of a blind attack, with no feedback from the attacked application and with ASLR activated at the same time, it is impossible to guess the exact address of our destination in advance. Entries like GOT are mainly at constant addresses but, as mentioned in Section 2.4, the compiler flag RELRO will protect this locations from write access. In our approach we will only write to the stack, which is always writeable, to change the execution flow of the application. A generic way of controlling the execution flow is to overwrite the saved instruction pointer on the stack, so that an address gets executed on a *ret* command that was chosen by an attacker. As we already described above, the stack frames are connected with a linked list with directed pointers. Our goal is to control the pointers in a way, so that we can write to arbitrary locations on the stack.

We will now describe our mechanism in more detail. To illustrate our mechanism, first imagine a chain of three function calls like shown in Figure 4.1. In this example a function `handle` calls a vulnerable function `parse` which in turn forwards the attackers buffer to an internal log function wrapper `logfunc`. This is a common scenario in both the Linux kernel and userspace applications. The initial stack layout of this scenario is depicted in Figure 4.2(a). If we consider a format string like %6$*hhn*, we will write to the destination of the 6th value on the stack. The number six would be the *offset* in our explanation. The size is given as multiples of the architecture size, in our case 32 bit. *EBP* is the saved extended base pointer of the calling function. We do not have to care about the stack cookie protection, but if there is a cookie it will be at the bottom of the box, which we assume in our case as part of the frame content like the used stack variables. As the stack is growing to lower addresses, it is possible to overwrite the contents of the stack frames of the function `handle` and `parse` from within the log function. The attack consists of three format string overwrites that use the pointers in *EBP*.

Note that the linked list of saved frame pointers is corrupted by this attack. An attacker may, nevertheless, reconstruct it after he is able to execute his own code, if it is required. This is only the case if the function is using local variables after the overwrite and before the return. Otherwise the application flow is changed and the attacker succeeded.



(a) Initial Stack configuration with three functions.

(b) In the first step, the EBP of handle() is redirected to the EBP of parse().

(c) In the second step, the EBP of parse() is redirected to the EIP of handle().

(d) In the third step, the EIP of handle() is redirected to the attackers code.

Figure 4.2: Sequence of overwrites to modify the return address

In the first overwrite, the saved *EBP* of the function `handle` (1) will be modified to point to the saved *EBP* of the function `parse`. This is achieved by using the offset X in the format specifier and change the content at offset $X + Y$ with the address of offset X. Now we can directly address the saved *EBP* of `parse`, as shown in Figure 4.2(b). The next overwrite then replaces the contents of the saved *EBP* of `parse`, located at offset X (2), to the *EIP* of the function `handle`, located at offset $X + Y + 1$, by using the offset $X + Y$, as shown in Figure 4.2(b). As a result of these first two steps an attacker generated a pointer on the stack, which points to the return address. In the third step the attacker overwrites this return address using offset X (3) to either point to the shellcode or some ROP chain, which the attacker prepared in advance. This final step is depicted in Figure 4.2(c). After the vulnerable function finishes, the control flow will switch to a sequence of instructions that was chosen by the attacker. An overview over the conducted overwrites is given in Table 4.2.

	offset in format string	dereferenced offset	value written (address of)
1	X	X+Y	X
2	X+Y	X	X+Y+1
3	X	X+Y+1	Address of ROP gadget

Table 4.2: Overview of required overwrites.

4.2.4 Pointer modification with ASLR enabled

In our approach we leverage the saved frame pointer feature as it contains pointers that can be used during an FSA. In case the attacked system has stack ASLR enabled, an attacker is unable to guess the address he has to write to the stack. Unfortunately in its simple version, ASLR is not randomizing the whole address. For example all addresses inside a page will be constant, as ASLR only randomizes the beginning of the stack on page granularity. This means that effectively the least significant 12 bits, we assume a page size of $4K$ as described in Section 2.4, of the address will be constant and not randomized. In the case of an FSA an attacker can benefit from this behaviour as he only needs to overwrite the least significant bytes of the frame pointer and redirect it to another frame pointer. Thus he modifies the least significant bytes of an address that is already pointing to the right location. Depending on the frame size the attacker has to overwrite one or two bytes. In the case of a good alignment and a distance less than 256 Bytes, an attacker does not need to care about ASLR, because only one byte write is required. We can only write a multiple of eight bits using the $\%n$ operator. This means that in case of a two byte overwrite, four bits of randomization are overwritten by the FSA. In a practical exploit this is not a problem and an attacker is able to brute force these four bits because of two reasons. First, if we have a network related application, each connection is transferred into its own process. This feature can be leveraged in a way that the attacker is able to crash the process without crashing the whole application. In case the exploit is not successful, the attacker can simply reconnect and try again. The second reason is, that only four bits of ASLR randomization is not a barrier for an attacker in this case, because the value only has to be found once. Every other write will also be at the same randomized four bits and can thus be calculated beforehand. Note that as the connection handler is forked for every connection the stack will also be at the same address until the main application is restarted. In contrast to the 12 bit ASLR randomization, some systems use 20 bits of randomization for the stack. In this case an attacker has to bruteforce more bits, but as we will show this case in our Proof Of Concept (POC), even with 20 bits of randomization the attack is practicable in a short time.

4.3 Proof of concept

After we have introduced a novel technique to change the control flow of an application in a blind way using an FSA, we now introduce our POC implementation. In the following we assume the attack to be conducted on a 32 bit Linux system on the x86 architecture. In our tests we used an Ubuntu 14.10 system with the latest security patches applied. Therefore, we assume that our binary is compiled with *gcc* in version 4.8. As already described, our vulnerable application consists of a networking daemon that forks a new process once it receives a new network connection. Each connection is than handled inside the newly created process. Our test system has the following protections activated: ASLR for stack, heap and libraries, NX on stack and heap, and RELRO. The stack addresses, where the EBPs are stored is randomized with 20 bit. After a local analysis of the attacked binary we will get the following values for the stack frame sizes: $X = 48$ Bytes and $Y = 48$ Bytes. This means that we will only require a one byte write if the least significant byte (LSByte) of the saved EBP of *handle()* is between $0x60 (= X + Y)$ and $0xfc (= 0x100 - 4)$. Otherwise it has to be a two byte write. As it is the more complex case, we will only consider the case of two byte writes and show, that this technique is feasible even with bigger frame sizes.

First of all, we will start with a simple bruteforce using three phases: In Phase 1, we will iterate over all possible values for the LSByte and restore the saved EBP of *handle()*. Since the addresses on the stack are 32bit aligned, there are only 64 possible values for the LSByte. If the value that is currently checked does not match, we assume that the application crashes and the server socket is closed. We can recognize this behaviour once we do not get any feedback. On the other hand we also have to take into account that not all successful tries imply a correct guess of the correct LSByte value. Thus after this step there can still be a number of false positives that we have to filter in the next step. Therefore, we will collect all checked LSByte values that do not crash the server immediately in the first phase and verify them in the second phase.

In the second phase, we reduce the number of possible values that we received in the first phase by verifying the integrity of those values. In our POC we designed four different verification tests that can be divided into two category. In the first category, we try to rewrite pointers on the stack by building a chained list of pointers. For example, as we know the stack layout, we can calculate the relative addresses of the saved frame pointers of other frames or any other variables within the frames and to overwrite their contents. In this case, we do not expect the application to crash. In the second category we also use those addresses where we assume pointers on the stack and redirect the pointer chain to point to a non-mapped memory location at the end, so `printf()` will crash during the memory write. After this verification process, we have the exact address of the LSByte.

The third phase is then required to obtain the value of the second byte. Thus this phase is only needed if we have a two byte write. In this phase we are writing to the second byte, which has 256 possible values in total. Since we now modify the saved EBP by a multiple of $4K$, the probability of having false positives is small. In our POC we did not get any false positives during our experiments. Our attack thus requires $256 + 64 + \delta$[1] connections in the worst case, which only takes few seconds in total. As the exploit strings used in this phase are small and can thus be executed very fast after `printf()` is called and the connections can also be multi-threaded, this step can be conducted in a short time.

After having the exact LSBytes of our address, we can now calculate all other stack addresses and build our exploit string to achieve an arbitrary write as describe in Section 4.2.3. The stack layout of our POC is illustrated in Figure 4.3, where every column represents the stack layout in one of the described three stages of our attack. In Stage 3, we overwrite the saved instruction pointer of `handle()` to return to a previously chosen destination. This destination could be the first ROP chain. Putting the whole ROP chain into the stack would assume that we have enough space on the stack for all gadgets. It would also require more space in the input buffer or many calls to `printf()` for many writes using the format string vulnerability. Therefore the ROP chain should be located within the buffer itself and the number of the written gadgets by `printf()` should be small. It should only be used to switch the stack to the heap and to execute another ROP chain. But this technique has a big constraint. Since the *libc* is randomized, the non-randomized gadgets are only available in the text section. We cannot guarantee that we can find enough gadgets in the text section, especially if the binary is small. It has been proven that the libc gadgets are turing complete by Schacham [101], so we set our focus to use the libc gadgets here. As our technique is based on a remote connection, the Procedure Linkage Table (PLT) contains network related functions like *send()* and *recv()*. We are going to use this feature for constructing a memory leak and to extract the address of the libc back to the attacker. The addresses of the used library functions like *send()* are stored in the GOT at a constant and readable address. The call to a library function is done inside the text segment, which is not randomized. We can either call it by returning to the text segment or we can call it directly using the PLT entry, which is also on constant addresses. Overwriting the return value with *send@PLT* and leveraging the send function also requires that we know the value of *clientsocket*, to return the information to the right client. This value could be bruteforced, but in many cases it is stored on the stack. In our POC, for example, the clientsocket is a parameter of the *handle()* function. We are using a gadget to lift the stack to the position of clientsocket and return to *send@PLT* with the arguments *(clientsocket, send@GOT, 4, 0)*. This sends the address of *send@libc* to the attacker, who in the next step is able calculate all addresses inside the libc and build an exploit for a successful attack.

[1] δ = false positive count * 4 (# of verification tests)

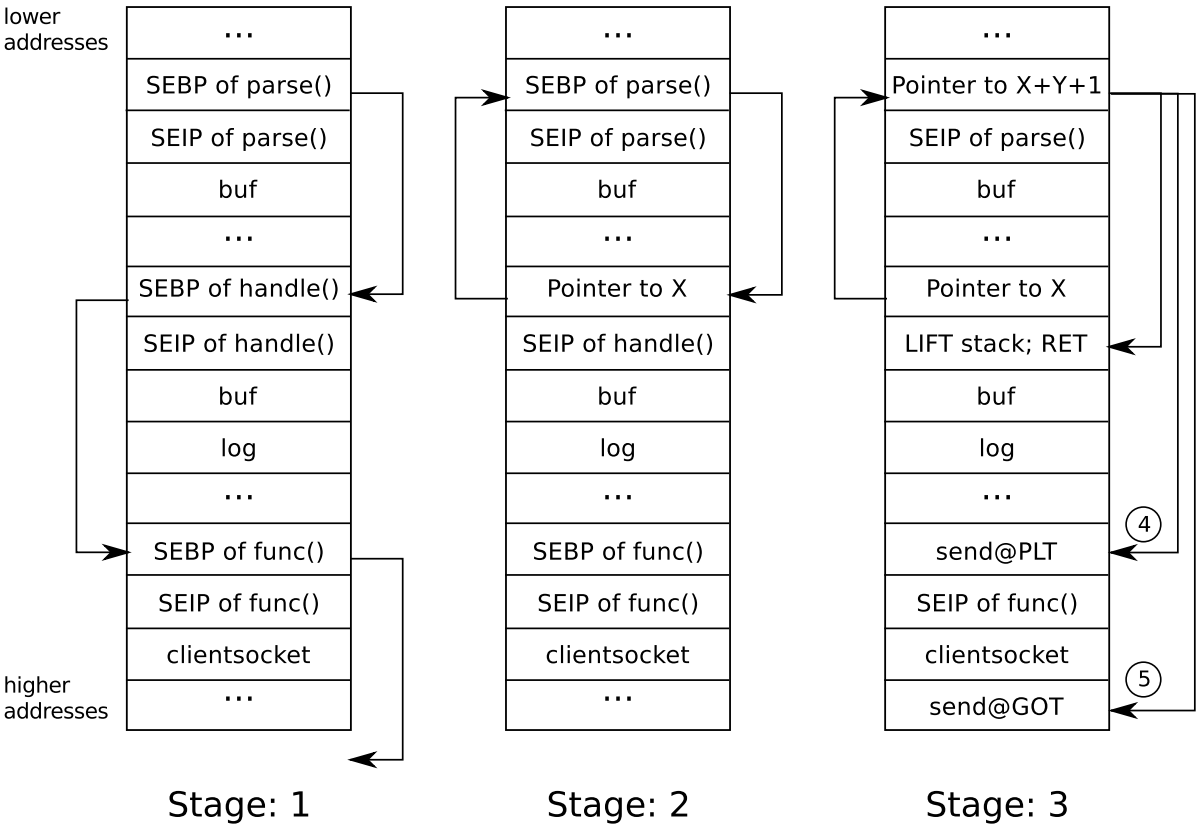


Figure 4.3: Stack layout for the proof of concept

4.4 Protection

The core of these attack is the presence of frame pointer, which are seen as a linked list. The frame pointers are modified to write to arbitrary locations on the stack. This mean, if we omit the frame pointers during the compilation process, there wouldn't be any pointer to be modified. Modern compilers are supporting this, e.g. in GCC we only have to add the flag $-fomit-frame-pointer$ in our compile parameters. If we do not have the source code, we implemented a tool called BinProtect [95] to protect binaries retrospectively. The approach is based on instrumentation techniques and can add additional code to validate the input before the vulnerable function is executed.

4.5 Summary

In this chapter we have shown that format string attacks are still a security issue in recent history. We proposed a new approach, which does not require a memory leakage

to exploit a format string vulnerability. Using our approach, we can exploit an FSA blindly without having any output channel to the attacker or access to the local system. Our concept extends the classical FSAs to write to arbitrary memory locations even in cases where the format string is not stored on the stack but instead resides on the heap. We especially show that it is possible to redirect the control flow of an application using and modifying only pointers that are already present on the stack. We have also considered the most known protection mechanisms like ASLR, NX, RELRO and have shown that blind format string attacks are feasible even with activated protections on the host. Finally, we proposed a way to protect against these attacks.

Chapter 5

Framework for Analysing Binary Applications

There is a vast amount of applications communicating over the network, for which the exchanged data is confidential and therefore encrypted. While encryptions helps to protect the confidentiality of the transmitted data, the application can still be vulnerable to remote attacks. For an intruder it does not matter if the transmission is encrypted or not, to exploit the vulnerable application.

Intrusion Detection Systems(IDSs) are used to detect such attacks. There are two categories of IDSs. First, a detection sensor located in the network, which is called Network based Intrusion Detection System (NIDS). Second, a local sensor residing on the same system as the monitored application. This sensor is called Host based Intrusion Detection System (HIDS).

When encryption is applied, the NIDS acts blindly and cannot protect against attacks, such as Blind Format String Attacks as described in Chapter 4. End-to-end encryption is designed to terminate at the destination application to fulfil the required security. By terminating the encryption at another node (e.g. a proxy server), the NIDS can inspect network data, but this will add an additional attack vector. So only a HIDS is capable to analyse these data, if we want to keep up the end-to-end encryption.

Beside this, the encryption layer makes it harder for the security analyst to test the remote application. From the point of view of a security analyst, a remote application can be tested by sending user controlled arbitrary data. The data is then parsed by the receiver. The problem here is that most vulnerabilities are located inside the application logic and can only be tested if the client/server is decrypting and parsing the user controlled data successful. To achieve this, one must rebuild an application for testing using the same algorithm and encryption key. Even this is already time consuming, but some applications have to reach a certain state of execution before the real vulnerability

can be triggered. For that purpose, more application logic has to be implemented by the analyst. A possibility is to combine the target application with a self-written application for the analysis. To achieve this, the analyst controls the target application and uses it to reach the potential vulnerable state. Afterwards, the key of the current state is extracted and passed to a self-written test application to send the arbitrary data. If the application updates the encryption key like in stream ciphers, we are locked out of the application and are not able to perform any actions using the original application.

In this chapter, we present our Intrusion Detection Framework for Encrypted Network Data (iDeFEND). iDeFEND is a generic framework to keep up the end-to-end encryption while still being capable to inspect plaintext data. We show the features of iDeFEND by describing two use cases for applications using encrypted network communication. As a first use case we present how we inspect plaintext network data and how we bridge the data for an IDS. This topic will be covered in detail together with the problems of a HIDS in Chapter 7. As a second use case we present a method to support analysts in testing network applications and to look for vulnerabilities. This use case will be covered in detail in Chapter 6. Our approach does not require any source code of the involved applications, nor the encryption key, nor information about the algorithm. iDeFEND is using a host based approach to solve the problem of encrypted network traffic. Instead of rebuilding the communication channel, we use the same channel of the application. We extract the information directly from the memory of the target application. This makes the whole encryption transparent from our view and we do not need to care about the encryption at all. We work at a layer above the encryption, where we monitor every traffic in plaintext. The evaluation is presented in detail in Chapter 9.

Parts of this chapter were published [64].

5.1 Framework Design

In this section we present the concept and the design of our framework for inspecting plaintext network data. If the whole network communication is encrypted, the application usually contains two wrapper functions. One is responsible for encrypting the plaintext and sending the data afterwards over the network. In our thesis we label this function Crypt and Send (CaS). The other function is responsible for receiving the network data and decrypting it afterwards to process the plaintext data. We label this function as Receive and Decrypt (RaD). These functions contain the plaintext data that is sent/received over the network. Both functions represent the core functionality for our framework. These functions are used in our use cases for extraction, interception and injection.

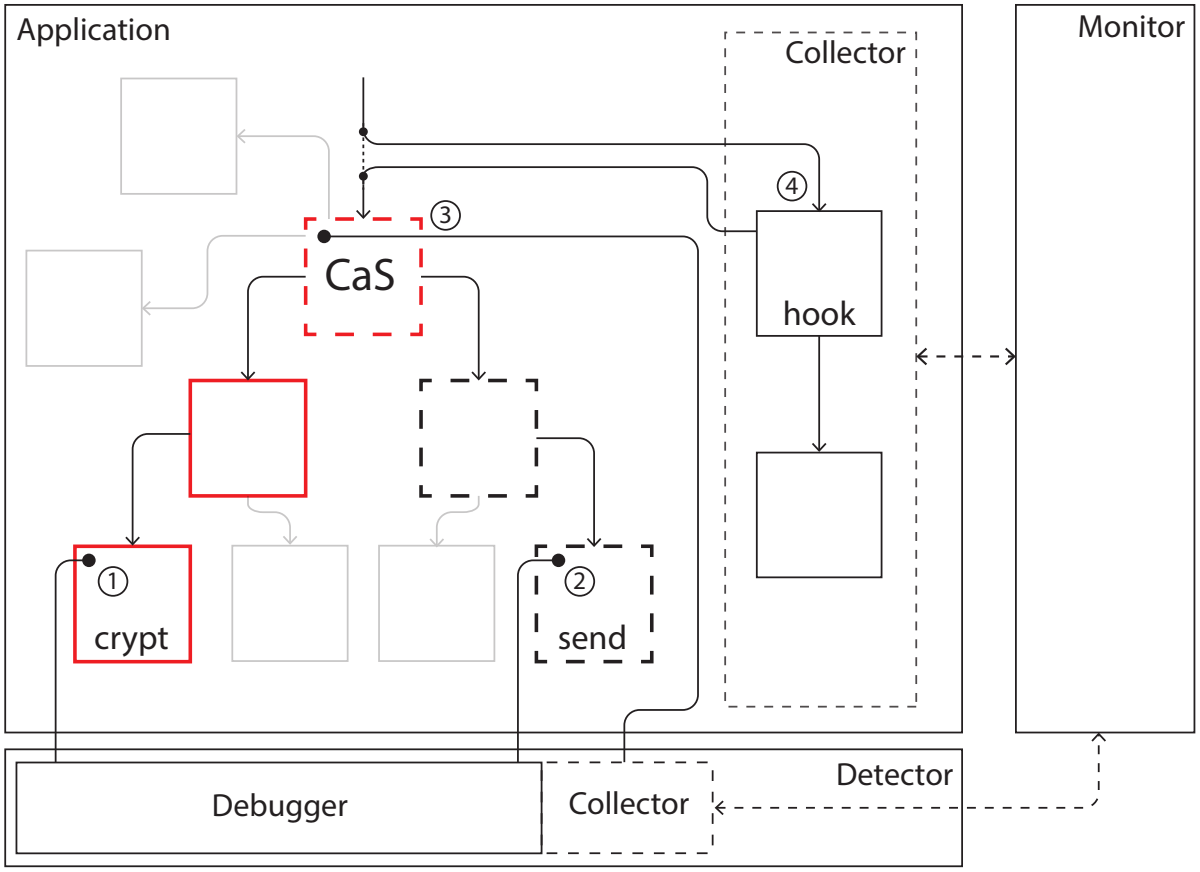


Figure 5.1: iDeFEND design

Figure 5.1 depicts the design of iDeFEND using the function CaS as an example. The case with RaD is analogous. Our framework consists of three parts: the *Detector*, the *Collector* and the *Monitor* modules. The *Detector* module is responsible for reverse engineering the offsets for the functions CaS and RaD. The application on the left side in Figure 5.1 contains the Control Flow Graph (CFG) with the CaS function in the centre. Tracking back the functions *crypt*(1) and *send*(2) using the underlying *Debugger* inside the *Detector* module, we have an intersection at CaS. The detection of the functions is described in detail in Section 5.2. The *Collector* module is responsible for gathering the plaintext data used for network communication from the application. This module requires the already identified offsets of CaS and RaD. iDeFEND supports two types of the *Collector*. One method is using the *Debugger* in the *Detector* module to directly extract or intercept/modify the plaintext data on CaS(3) or RaD before it is processed by the target application. The other method is placing the *Collector* directly into the process space of the application. This is setting a hook(4) on the function CaS and RaD to capture the information passed to these functions. Each time the application sends or receives encrypted network data, the *Collector* will gather the plaintext data. We pass the collected data to the *Monitor* module. This module is responsible for handling the plaintext data and for providing an interface to IDSs.

5.2 Function Identification using the Detector

In this section, we describe the *Detector* module of iDeFEND. Before extracting information from a process, we need to identify the application's CaS and RaD functions. To achieve this, we use the breakpoint features of a debugger. Since we have to deal with encrypted network traffic, the application has to provide at least three functions: *crypt*(1), *send*(2) and *receive*. Send and receive are in general the public library functions of the Operating System (OS), thus we can retrieve their address easily. The *crypt*(1) function is responsible for en-/decrypting the data. Depending on the algorithm it can be one or two functions. If a dynamically linked encryption library is used, we can identify the crypt offset easily by looking for the API export of the library in the memory. Otherwise, we have to identify the function inside the binary. There are already some approaches to detect functions, which are used for encryption [23] [51]. If they fail, we will use a more generic approach for function identification, which we describe in Chapter 8.

Having the offsets of *crypt*(1) and *send*(2) illustrated in Figure 5.1, we set hardware Breakpoints(BPs) on these functions and start the application. We are only interested in encrypted network traffic, so we have to make sure we do not catch too much data. In case of outgoing network data, the important plaintext is only the plaintext that is encrypted and sent afterwards. The constellation of *crypt*, *send* and CaS appears only if the application is the correct state. As an example, the encrypt function can also be used

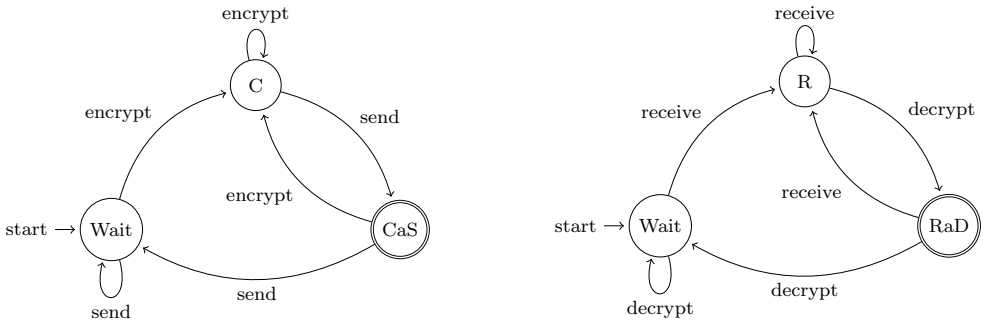


Figure 5.2: *Debugger* states

by other functionalities, e.g. encrypting a file. Figure 5.2 illustrates the possible states of the application depending on the occurrence of the BPs. After starting the application we wait until the first encryption BP occurs otherwise we stay in the same state (Wait). At this point, we are in the state *C*. We save the application state including all relevant data, such as registers, the memory content and the application stack. After resuming the application the next BP is either on *crypt*(1) or on *send*(2). At *crypt* the application is encrypting some data for sending or internal use. At *send*(2) the application is going to send some data and we change to the final state *CaS*. At this final state, there are only two possibilities. Either some plaintext data is transmitted over the network and the encryption was for internal use, or the encrypted data is transmitted over the network. We evaluate this by comparing the data modified by the encrypt function with the data accessed by the send function in state *C*. If they match, the same data is going to be send over the network. In that case, we can identify the CaS function by comparing the partly reconstructed CFG of both states with each other and look for an intersection as illustrated in Figure 5.1. We retrieve the execution flow by backtracking the caller functions. We aim for the intersection of both execution flows. This is most likely the CaS function. In the case of incoming network data, we have to detect the RaD functionality. This works analogous to the identification of CaS. We propose another approach in Chapter 6 if no RaD function is available in the application.

5.3 Information Extraction using the Collector

In this section, we describe the methods to extract the necessary information from the detected CaS and RaD functions to pass it to the *Monitor* module. This feature is supporting the use case of inspecting plaintext network data to identify intruders. The method is describing the module *Collector* in iDeFEND. Our framework supports two ways for extraction.

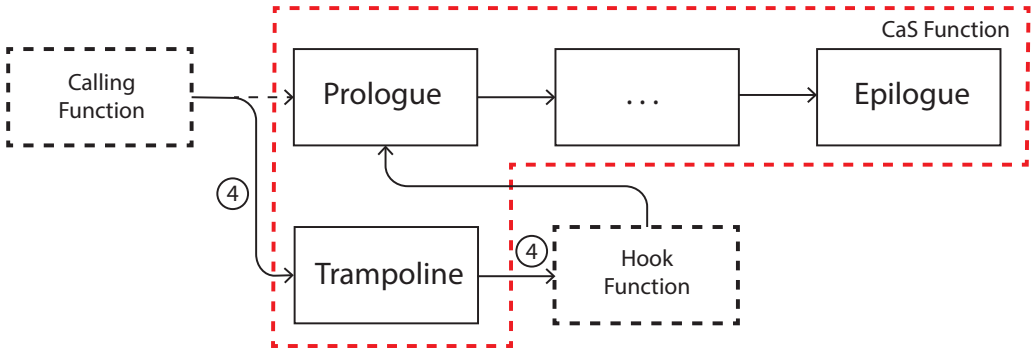


Figure 5.3: Function Hooking

First we describe the *Collector* module using the *Debugger* illustrated at the bottom in Figure 5.1. We place one BP at the entry of CaS(3) and another one inside the RaD. The plaintext of the encrypted messages is passed to the function CaS as function parameter. When the application halts at CaS(3), we retrieve the data from the parameters. In the case of RaD the encrypted message is decrypted and parsed afterwards. To extract this data we set the BP immediately after the decryption is done. When the application halts at RaD, we extract the plaintext either from the return value or the modified input parameters. The location of the decrypted data is already identified by the *Detector* module. The next step is to pass it to the *Monitor* module to analyse it.

Our second method uses the *Collector* module inside the target application, illustrated on the right side of the application in Figure 5.1. Instead of exits to the debugger we run our own code inside the target process. For this purpose, we inject our own module into the target application. Figure 5.3 shows how we use a trampoline to redirect the detected functions to a place inside our module. To achieve this, we replace the initial prologue of the detected functions with a *jmp* instruction. This redirects the program to our code. In our additional code we save the input parameters to the original function. We restore the original prologue of the hooked function, call it with the original parameters and save the return value. Another thread collects the extracted data and delivers it to the *Monitor* module. Our function returns during this information exchange and the applications resumes as intended.

5.4 Packet Injection and Interception

In this section we consider the second use case and describe the method for sending user controlled arbitrary data using the application's CaS function. To achieve this, we use an equal technique as described for the *Collector* module in Section 5.3 to load our own module *Injector* into the target application. Having our module inside the

target application, we build an Inter-Process Communication (IPC) to exchange data with the module. To send arbitrary data to the server, we use this channel to pass our plaintext network data to the Injector. As being part of the application, we call the CaS function directly with our input as parameter and let the application do all the necessary data modification, like splitting, encrypting and sending. As IPC channel for communicating with CaS, we use code injection to place our precompiled calling stub into the application. This stub will get the objects for sending directly from the new allocated memory inside the application and call the CaS. The benefit here is, that we do not have to care about any algorithm or the encryption key. We also do not have to care about the states the encryption algorithm proposes as in stream ciphers. iDeFEND is also able to intercept and modify the plaintext data transmitted over the network. We use the same hooking technique as described for the *Collector* module in Section 5.3. Each time a packet is sent or received our hooked function retrieves the data. We use the IPC to interact with the security analyst using an external graphical user interface. At this point, the application is halted and the tester inspects or alters the data for testing purposes.

5.5 Implementation for the x86 Architecture

We have implemented a prototype of our framework. Our implementation was running on a machine with an Intel Core i7-4600U CPU 2.10GHz CPU and 8GB RAM. We used two virtual machines with Windows 7 Professional with Service Pack 1 and Ubuntu 14.04 LTS as OS. The *Detector* module uses a self-written debugger to place and handle hardware BPs inside the target application. We set two hardware BPs on *crypt* and *send* to detect CaS, RaD is detected analogously. The *Collector* inside the *Detector* module places a BP at the entry of CaS(3) to extract the parameters given to the function and another one inside the RaD to have access to the already decrypted data.

When the application halts at one BP, we gather the memory pointer directly from the stack and registers. After dereferencing the pointer we extract the plaintext data out of the memory. Since this technique stops the application for all incoming and outgoing network packets, the application slows down. To avoid this, we implemented another method to extract the data without incurring performance. We execute our code directly inside the target process space. The benefit in doing so is, that we sustain effectiveness and do not delay the target application during the information extraction. Executing additional code in another process is easily achieved using module injection. This allows us to provide code fragments or even functions in the target application.

Our framework supports two methods to inject additional code into the target process. On Windows OS we use the default debugging API calls to load our Dynamic Link Library (DLL) into the application. The second method can be used for Windows and

Linux OSs. We scan for unused memory inside the binary, which is called code cave. We use the code cave to insert the code snippet for loading the module and execute it by manually setting the instruction pointer (IP) of the application [5]. After the injection of our module, the hooks to the CaS and RaD functions are placed. Our *Collector* logs the parameters and return values of the hooked functions and passes them to the *Monitor* module. By default iDeFEND performs this by using a IPC with the *Monitor* module running on the same system. The modules do not have to run together on the same machine. The *Collector* module of iDeFEND can also be used separately. We set up a clean test system to use the *Detector* module and to generate the relevant offsets for CaS and RaD. This data is written into a config file and the *Collector* module loads the offsets to attach to the productive system and start extracting the plaintext data. iDeFEND also allows to send the gathered data from the *Collector* module over the network to run the *Monitor* module and an IDS on a different machine.

5.6 Summary

In this chapter, we proposed iDeFEND, a framework to analyse the payload in encrypted network communication. We have shown how we can inspect the encrypted network data of closed source applications in a use case. Our method can automatically identify the related functions for encrypted network communication inside applications. We made use of the identified functions to extract plaintext network data from the application using the *Collector* module. We used the collected data for further analysis in our *Monitor* module to detect exploits. With iDeFEND, we can use the identified functions to intercept and modify the current plaintext network data to change the parameters sent to the target application without the need for reverse engineering of the encryption algorithm and key. We showed how to use the identified functions to inject arbitrary data and enforced an unintended data transmission.

Chapter 6

Security Testing of Mobile Applications

Encrypting the network traffic prevents attackers from accessing sensitive data, but cannot stop them from exploiting security flaws in the implementation to achieve crashes, intrusion or code execution on the system. Security testing is responsible for detecting these vulnerabilities at an early stage. However, even powerful testing frameworks are blind when end-to-end encryption is applied and can only randomly generate or mutate packets. Additionally, the encryption layer makes it difficult for security testers to validate the remote program which increases the risk of missing faults. Solutions to this issue usually require a high amount of reverse engineering, since most of the target applications are closed source.

iDeFEND was implemented and evaluated for the x86 architecture, but nowadays most of the networking applications are running on mobile devices like smart phones, tablets or other embedded devices whose processors are primarily designed by ARM. Since the framework uses hardware dependant features, its concept must be adapted to the specifics of the new platform.

Additionally, mobile applications tend to buffer network packets in a queue before sending them. This compensates bad connectivity, but results in a conflict with the current approach of iDeFEND. Furthermore, the framework relies on the presence of a specific wrapper function to inspect the received, unencrypted network data. In practice, this function can be more complex than expected by the framework and requires additional reverse engineering.

We overcome these shortcomings and extend the iDeFEND system. We provide a framework that allows to use common security testing tools for encrypted network applications. The evaluation is presented in detail in Chapter 9.

Parts of this chapter were published [65].

6.1 Testing Applications using iDeFEND

In this section we present a use case of the iDeFEND framework and explain how it enables security testing of encrypted network applications.

The iDeFEND framework is designed to support security testing of proprietary, closed source software. This type of testing is referred to as black box testing, since we examine the functionality of the programs under test without knowing details on the development, program internals or implementation. Even though the program is a blackbox, security analysts are still able to use powerful fuzzing tools to test for commonly known vulnerabilities. They can, for example, test a server against blind format string attacks as described in Chapter 4. In this scenario, a security analyst sends attack strings to the server application, lets it interpret the data and afterwards validates the response and thereby, the outcome of the test case. Since no information about implementation and design of the target application is available, also the internals of the encryption are unknown. This means, the analyst does not have information about the encryption algorithm, the encryption key or encryption protocol. As a result, the test messages of the security analyst cannot get past the encryption layer and thus, program internals cannot be tested appropriately. Either the plaintext test message does not fulfil the specification of the protocol which leads to rejection of the data. Or the test data is accepted but expected to be encrypted and thereby, arbitrarily changed during decryption. Figure 6.1 illustrates this scenario with the orange arrow representing the test string data. The diverging arrow heads symbolize the arbitrarily changed data after decryption. Since the test data is changed, it will not trigger the functionality the tester originally intended to.

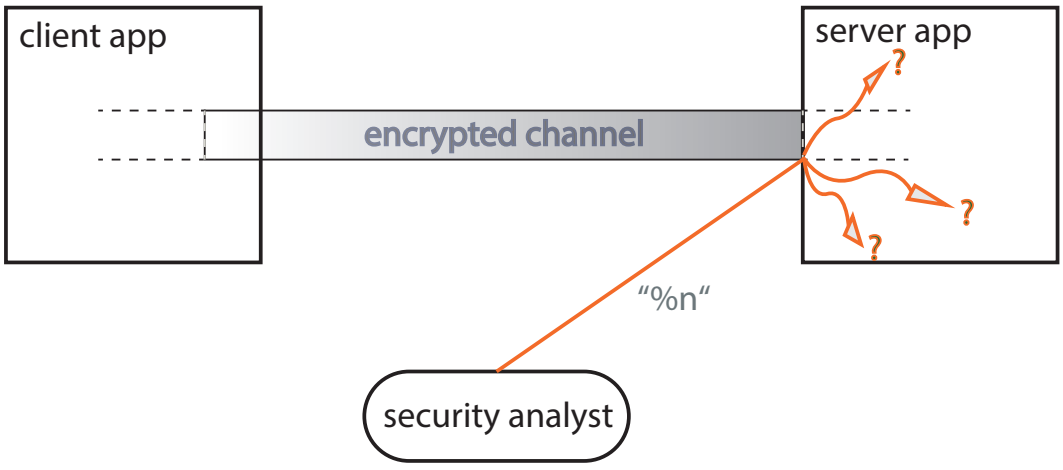


Figure 6.1: Security Testing of Encrypted Communications

If the security analyst wants to test the server application as intended, he can use the iDeFEND framework. Using the framework for testing circumvents the issue of encryption. It provides an interface for the security analyst to the client application and thus, access to the encrypted channel. This way, the security analyst can pass the plaintext test data to the framework interface which uses the client application to encrypt and send the data. The sent data then is decrypted correctly at the server application and eventually triggers the intended functionality. Figure 6.2 shows the flow of the plaintext test data with the green arrow. The security analyst passes the data to iDeFEND, which is using the wrapper function in the client application to inject the data into the encrypted channel. Since the wrapper function is handling the encryption and network communication, the test data is sent to the server application like any other program internal message and the test data is decrypted at the server correctly.

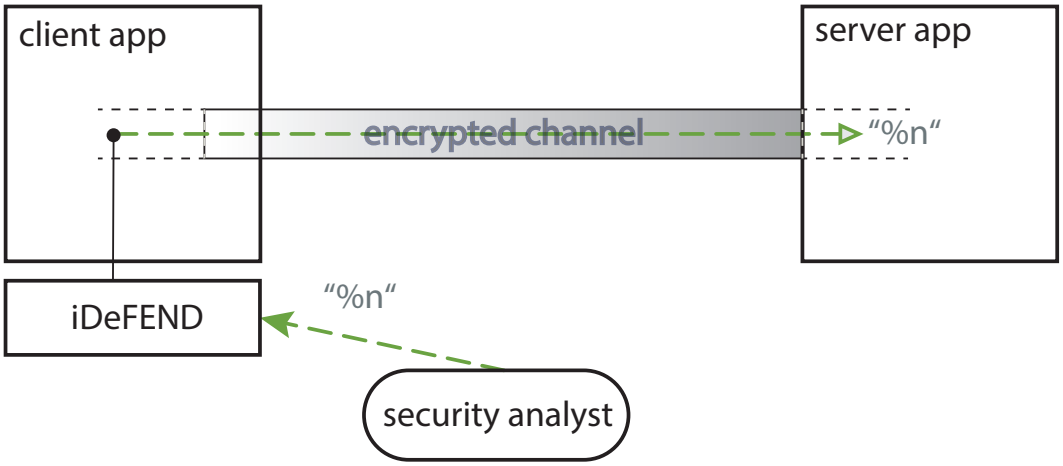


Figure 6.2: Security Testing of Encrypted Communications with iDeFEND

6.2 Improvements of iDeFEND

In this section we discuss the limitations of the current iDeFEND approach for software testing and present our improvements to make it applicable for a wider range of applications and test setups. We put focus on the conceptual weaknesses of the framework and separately address the transfer to ARM in the following section 6.3.

Currently, iDeFEND implements the identification of the wrapper functions with backtracking. Therefore, the call graphs at successive calls to the logic function pairs are intersected. Knowing, for example, that wrapper CaS is responsible for calling encrypt and send, means that the call graphs of encrypt and send must have an intersection at

the wrapper function. This approach introduces a weakness. The wrapper functions can only be detected when they successively call encrypt and send. For applications that use a message queue in network communication, this assumption is never met.

Additionally, iDeFEND defined the RaD wrapper function to return the decrypted plaintext packet. It inspects the plaintext data by hooking the function at its return instruction. This requires detailed knowledge about the structure of the RaD function and the presence of RaD function.

In the following subsections we propose solutions to those two problems.

6.2.1 Test Data Injection into Message Queues

In general, applications can implement encrypted network traffic in two different ways. Either data is encrypted and sent at the same point in the program, or at separate points. If the data is encrypted at one and sent at another point, the data must be stored in any kind of packet queue. This means that the program is encrypting the data, enqueuing it, later dequeuing and eventually sending it. In this scenario, the wrapper function is encrypting and enqueuing, instead of encrypting and sending data. This is an issue, since the wrapper function EnCrypt & EnQueue (CaQ) cannot be detected with the existing approach of iDeFEND. The current algorithm relies on knowing the addresses of both encrypt and enqueue. To our understanding, there is no generic way of identifying the address of such an enqueue function. This means, the current design of iDeFEND cannot be used for security testing of applications with packet queues.

We addressed this issue and analysed the structure of such applications and came up with a solution. Figure 6.3 illustrates the control flow graph for the wrapper function CaQ. Usually, programs implement protocols that construct different packets for many different purposes. This means that for each packet the wrapper function is called from a different calling context. Independent of the packet, the wrapper function is calling the same encryption routine. Projecting this to the call stacks at encrypt, independent of the packet type, all call stacks share the same function frames beginning at CaQ. For this reason, the CaQ function can be identified as soon as at least two call stacks from different calling contexts are collected.

Our solution to the issue of identifying the CaQ function is to record all call stacks at encrypt and intersect them to find the wrapper function. Going from top to bottom, the last function frame that is the same for all recorded call stacks, reveals the CaQ. In order to avoid call stacks that are a result of internal encryption, the encrypted data is validated to be network traffic as soon as it is send. Since the data is copied to the queue, the pointers at send and encrypt vary. We handle this problem by not saving the pointer itself, but the whole buffer. At the validation of the data flow we simply compare

the contents. When the data is validated to be network traffic, the corresponding call stack at encrypt can be used for the detection of CaQ.

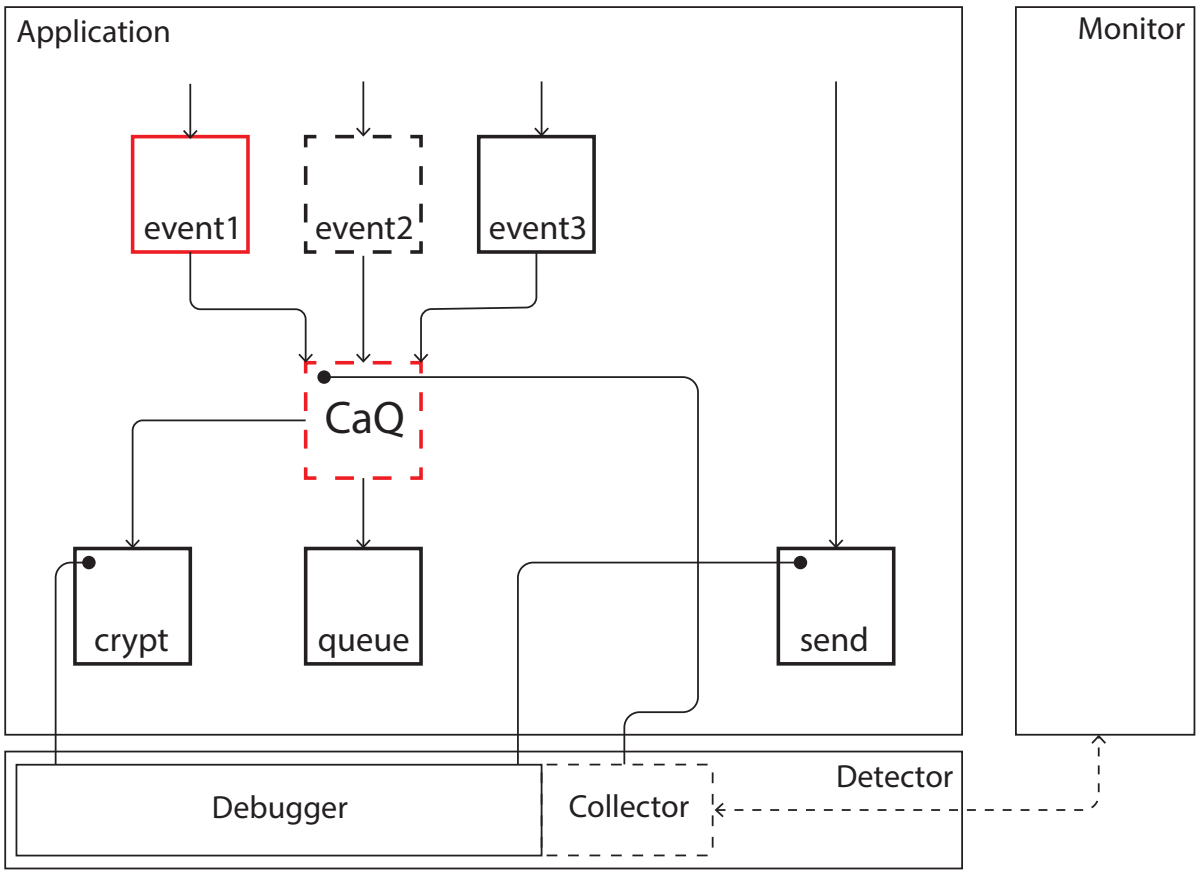


Figure 6.3: CFG for wrapper function CaQ

6.2.2 Generic Approach for Data Inspection

The second problem of iDeFEND is that the current approach assumes the existence of a specially structured RaD function, which is not always the case. The RaD is assumed to return the decrypted plaintext data. iDeFEND hooks the RaD at the return and extracts the plaintext data. However, many applications do not implement this type of wrapper function. In general the receiving wrapper function is a loop that never returns. As illustrated by figure 6.4, the RaD loop calls the *receive* function and passes the data to a parsing unit. The parser then decrypts the data. Without knowing the structure of the RaD, the current iDeFEND cannot inspect the plaintext data. The correct offset and the information about the correct register or data pointer have to be known at this point.

We analysed this issue and came up with generic solution. We do not rely on the presence and detection of wrapper functions. Our generic solution works for applications that only implement the basic functions (de-)crypt, send and receive. Additionally, our improved approach does not even rely on frame pointers.

Similar to the original approach we also break on receive and decrypt. However, we identify data that is received from the network not by comparing the pointers of data, but by comparing the content of input and output buffer between receive and decrypt. The idea is the same as it was for validating data that is going to be send over the network for the CaQ. When the decrypt function returns and we validated that the encrypted data was received from the network previously, we extract the plaintext data from the returned buffer. The extracted data then can be passed to the tester for inspection.

With this method we extended iDeFEND to allow the inspection of unencrypted server responses, even though the application does not implement a wrapper function and use frame pointers.

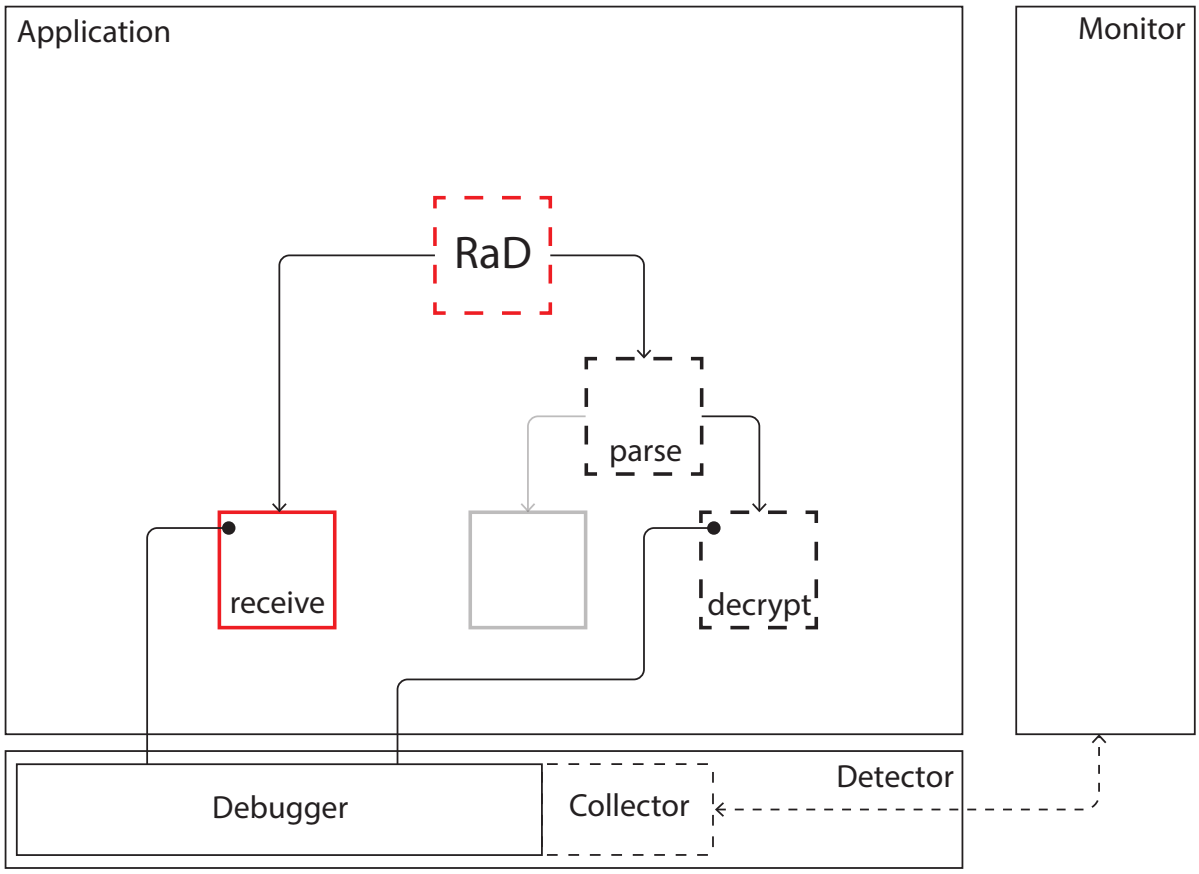


Figure 6.4: CFG for the function RaD

6.3 Transfer to the ARM Architecture

In this section we discuss the transfer of the iDeFEND framework to the ARM platform. We present the key differences between the x86 and ARM architecture and describe how the concept can be applied to the new environment.

In the security testing scenario presented in 6.1, the client application is usually running on a mobile or embedded device. Most of these devices use processors whose design is based on the ARM architecture. The current concept of iDeFEND is only applicable for x86 and thus, does not support ARM devices. We transferred the current concept of iDeFEND to the ARM architecture. The key features that had to be transferred are debugging with hardware breakpoints, data extraction at function calls, call graph reconstruction from the stack and hooking of functions on machine code level.

6.3.1 Using Hardware Breakpoints for Debugging

Both architectures x86 and ARM implement both hardware and software breakpoints. Hardware breakpoints offer a better performance, do not require modification of the executable code and thus, are less obvious to detect. This makes them perfectly suited for implementing the detector module of iDeFEND. They are limited to a small number of breakpoints per processor, but that is unproblematic, since the specification of x86 offers up to four and ARM offers up to 16 hardware breakpoints. For the implementation of iDeFEND at most four breakpoints are necessary to trap on the target procedures send, receive, encrypt and decrypt.

On x86, one debug register is responsible for one breakpoint. It holds the address or value depending on the type of the breakpoint. A shared control register is responsible for all debug registers. It holds flags to enable, disable and configure each breakpoint. On ARM, each hardware breakpoint consists of two registers, a control and a value register [76]. Since iDeFEND only requires the insertion of breakpoints on instructions, which means addresses, the breakpoint value register is simply set to one of the addresses of the target functions. The control register has 32 bits with flags for several options that allow, for example, to link different breakpoints, enable or disable them, specify the privilege and exception level the breakpoint debug event is generated on. In our use case we want to break on application level and only on address matches. Obviously, this means that for the control register the enable flag must be set. The breakpoints are not required to be linked and should only trigger on an address match. Removing a breakpoint is simply done by flipping the enable-bit in the control register.

6.3.2 Extracting Data from Procedure Calls

The next important feature that requires adaptation to the ARM architecture is the extraction of arguments on procedure calls and thus, the extraction of data pointers of in- and output when one of our breakpoints is triggered. Extracting the data is necessary, for example, to detect and exclude internal encryption and to gather data with the collector module. Since the breakpoints are placed on the prologues of the target procedures, which means on the first instruction of the procedure, the function parameters are still untouched and remain in the same registers and stack position as they were passed by the calling procedure. This means, the function parameters can be accessed based on the specification of the underlying calling convention.

ARM, in contrast to x86, defined its own procedure call standard [78] and recommended its usage. On ARM, the first four parameters are always passed in the in the first four registers `R0` to `R3`. Every additional parameter is pushed to the stack. Since the *Stack Pointer* register always holds the address of the top of the stack, the arguments five and higher can be accessed by a simple read from the stack pointer register and accessing the memory at the obtained address plus the argument offset.

It is, however, not always the case that all input and output buffers and buffer sizes are passed as function parameters. Depending on the implementation of the target application, the encrypt function, for example, may also use the return value to return the address of the ciphertext buffer to the calling function. In this scenario, it is necessary for the debugger to resume the application and break again on the return of the procedure, in order to extract the return value.

On x86 return values are typically placed in the `EAX` register. However, breaking on the return address of a procedure on x86 is complex, as there usually is not only one but multiple return instructions for different control flows. That means it is necessary to break on all returns within the procedure to be sure to get the correct return, and then extract the return value from the register when one of them triggers. Depending on the amount of available hardware breakpoints this is probably not feasible. On ARM this problem is not present, since each processor has a special *Link Register* that always holds the return address of the current function call. It is easy to place a breakpoint at the return address and wait for the procedure to finish. When the breakpoint on the return address triggers, the return value can be extracted from the `R0` register.

6.3.3 Call Graph Reconstruction

In order to detect the wrapper functions CaS and RaD in memory, iDeFEND proposes to reconstruct and intersect the call graphs for the function pairs encrypt and send, and receive and decrypt. The call graph in this case represents the different levels of

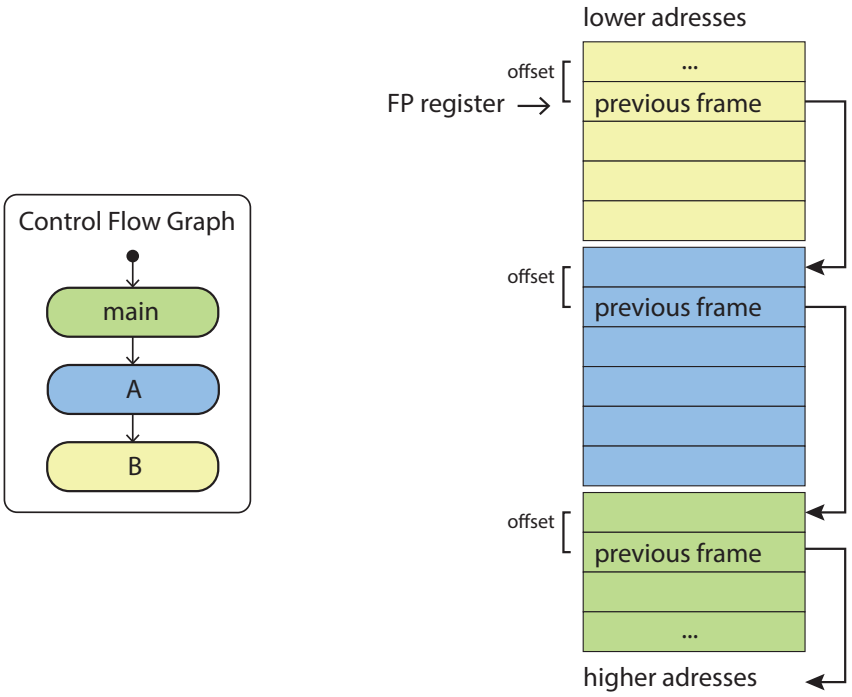


Figure 6.5: Stack layout on ARM

procedure calls. For example, when the main method of an application calls a function A which uses a function B and where our breakpoint was placed on B, then the call graph at B would consist of the three procedures main, A and B (see Figure 6.5). On the lowest layers of software, procedure calls are managed by the stack.

The stack exists on both x86 and ARM, and is split into multiple stack frames, whereas each frame represents a procedure call level in the call graph. When a procedure is called, a new frame is pushed to the stack, and when this procedure finishes and returns, the frame is popped again. Based on this design, the reconstruction of the call graph boils down to the reconstruction of stack frames from an existing stack. On x86, the unwinding of the stack frames is quite simple. The register `EBP` holds the address of the current stack frame. On every procedure call, the previous value of `EBP` is pushed to the stack and `EBP` is updated to the current stack pointer. The reversal of this process, which is required to unwind the stack, is as simple as reading the address from `EBP` to obtain the memory address that holds the previous frame pointer. This value then can be used to get the frame pointer before and so on until the bottom of the stack is reached.

On ARM, unwinding the stack is a more complex task. In general, the architecture does not provide a dedicated frame pointer register. Depending on the optimization level of the underlying compiler, stack pointers might not even be present on the stack. This is problematic, since it is highly complex to reconstruct stack frames without having

frame pointers, as it requires a sophisticated analysis of the stack. Nevertheless, there are other methods that can at least reconstruct parts of the call graph, which are sufficient to detect the wrapper functions. One idea is to use the `LR` register to keep track of the procedure calls with the debugger at runtime. Since the register holds the return address of the current procedure call, it is possible to step back to its caller with a single breakpoint. This means, it is possible to go through the call graph backwardly, which in our case is sufficient for the detector module. The idea is to insert breakpoints on the four target functions send, receive, encrypt and decrypt. Consequently, the approach to detect the wrapper functions is to wait for a break on encrypt or receive and then step backwards until the breakpoint on send triggers or decrypt, respectively. When this is the case, the previous break must have been inside the wrapper routine. In case, where during this backwards stepping any other breakpoint than the expected triggers, the call cannot originate from the wrapper function and the stepping must be aborted. This approach requires an additional hardware breakpoint, is slower based on potentially many application interrupts and also needs a disassembling logic to find the start address of the wrapper functions, since we end up anywhere inside the function when simply breaking on return addresses.

A faster and simpler to method is based on the functionality of compilers. They allow to have a properly set up stack with frames and a dedicated frame pointer register, depending on the configuration. This of course requires the target application to be build from source with the correct compiler configuration. Table 6.1 illustrates the effects of different flags, configurations and optimizations on the generation of stack frames for the GCC compiler. The flags *mapcs-frame* and *fno-omit-frame-pointer* force the compiler to preserve stack pointers throughout all optimization levels. The only difference is that the pointer offsets vary. Without them, the compiler only generates stack pointers for optimization level O0, which means no optimization. As a result, closed source software which was compiled without additional flags but optimization enabled, do not have stack pointers and thus, are not supporting the call graph reconstructing described earlier. Compilations that do not specify any additional flags or optimization have frame pointers.

Additionally, it may occur that the wrapper functions call libraries which redirect the call to send, receive, crypt and decrypt. Normally, this should never be the case, but when it is, all of the libraries in between must be also compiled with stack pointers enabled.

6.3.4 Hooking Functions

In case the collector module uses module injection for the extraction of plaintext data from the target application, function hooking becomes a requirement. The idea behind hooking is that each time the target function is about to be called, the execution is

GCC Flag	Optimization				Offset to next frame pointer (FP)
	O0	O1	O2	O3	
no flags	✓				FP - 4
-mapcs-frame	✓	✓	✓	✓	FP - 12
-fno-omit-frame-pointer	✓	✓	✓	✓	FP - 4
-mapcs-frame fno-omit-frame-pointer	✓	✓	✓	✓	FP - 12

Table 6.1: Presence of Stack Pointers with different Compiler Settings

redirected to a function in our injected module that contains our own implementation of the function. This redirection of control flow is implemented on assembler level by modifying instructions in the executable section of the application. In order to guarantee that the hook is always executed when the function is called, the trampoline, the code that redirects control flow, must be inserted at the beginning of a function.

On x86, the prologue of each procedure consists of three instructions that together have a size of six bytes. Since those instructions do neither modify the program counter nor contain program counter relative offsets, they are address independent and thus, can be moved to a different location in memory. This means, that six bytes are available at each function to insert, for example, a jump instruction which redirects the execution to another function. In order to not crash the program, the function in the module must save all registers and execute the three missing instructions before returning to the original function. Since x86 has instructions with variable lengths, only instructions of a size smaller or equal to six bytes can be used. If the instruction is smaller, it simply can be padded with `NOP` instructions that have one byte and keep the processor idling. Six bytes is enough to insert either a jump which is relative to the current instruction pointer or a jump that uses an absolute address from memory. For relative jumps, the code that is placing the hook also has to calculate the correct relative offsets at runtime. Using the call instruction as a trampoline is a bit more difficult, since it does not only branch but also push the return address to the stack and thus, complicates, for example, the access of function parameters.

On ARM, the function hooking must be handled differently, mainly based on the heavily varying instruction sets. Here, instructions have a fixed length of four bytes, which makes substitution of instructions simple. The prologue of procedures, however, is not always the same as it is on x86. The first instruction can be memory address dependent or independent. In the worst case the first instruction is address dependent and cannot be moved, as it contains a program counter relative instruction, which frequently is the case in loads from constant pools. There, hooking is not possible. In the other case, where the instruction is a move or a push to the stack, the instruction is address independent and can be moved. ARM has an unconditional branch instruction that is suitable to

insert hooks, as it jumps to a memory address relative to the current program counter. The range, however, is limited to an offset of 26 bit, which can be problematic in many scenarios. On Linux, for example, the code of the base module, the application itself, is mapped to the lowest addresses in memory. Shared objects as the injected module are mapped to the highest addresses. This results into a gap, which is significantly larger than the 26 bit can specify.

Another possibility to branch, and solution to the range problem, is to directly modify the program counter. Here, the limitations are that the prologue offers only space for one instruction. ARM does not provide 32 bit constant moves, but only an instruction that loads a 32 bit value from memory to a register, relative to the current program counter. This means, it is possible to substitute the first instruction of the target procedure with this load and perform the branch. The memory location that holds the address must be close to the load instruction and can only be safely placed on dead code. Since compilers always use multiple bytes of padding between two procedures in memory, the padding would be a good location to place the address.

6.4 Implementation

We implemented the improved iDeFEND framework on an ARM device that is running a Linux operating system. Choosing Linux for this task seemed reasonable, as most of the target ARM devices like smart phones, tablets or embedded boards are either running Linux or Android, which is also based on the Linux Kernel. We decided to use a Raspberry Pi 2 embedded board that is equipped with a 900MHz quad core ARM Cortex-A7 processor and 1GB RAM. It uses the Linux distribution Raspbian 4.1.13-v7 as operating system.

We split the implementation into two parts. First we present the detector module, followed by the implementation of the collector module.

6.4.1 The Detector Module

The base module of iDeFEND, the detector, is a debugger which is specifically geared towards detecting the addresses of the CaS and RaD functions in the target, remote program. On Linux, application level debugging is provided by an API called *ptrace*. It has a very generic interface that abstracts the access to the hardware. Since ptrace can only attach to single threads, and that we want to debug the whole processes, we attach it to every thread of our target process. In order to get the required interrupts on send, receive, encrypt and decrypt, we insert hardware breakpoints on these four functions. This is implemented with a ptrace call that writes to the breakpoint value and control

registers. Since the Raspberry 2 provides five hardware breakpoints, we insert all four breakpoints at once. The control registers are set to `0x1E1` and the value registers to the address of the target functions. The control register value is composed by the enable flag and the flag to only break on an address match.

After attaching to the process and inserting all hardware breakpoints, the debugger iterates through a loop that waits for debug events. This is implemented with an API call to *waitpid* with the *WALL* flag set, which results into a blocking wait for any debug event from any thread of the remote process. When a breakpoint is hit, the remote program is analysed and resumed afterwards. Since breakpoint exceptions force the processor to not increment the program counter, simply resuming the paused thread would again trigger the same breakpoint. Therefore, we move the breakpoint to the next instruction, which on ARM is always four bytes later, resume and wait for the next break to move the breakpoint back to its original location. This single step only leads to correct program behaviour when the skipped instruction is not a branch or program counter modifying operation. This is guaranteed, as we only break on function prologues.

6.4.1.1 Finding addresses of Send and Receive

In order to insert the breakpoints at the correct locations, the virtual addresses of the target functions are required. iDeFEND expects that the encrypt and decrypt function addresses are already given by either having access to the source code or an external detection tool. The remaining two functions send and receive will eventually be exported functions of the Linux system library. This is based on the fact that network communication is also abstracted by the operating system and thereby, applications will, at least at the last instance of the call, use the system library. On Windows or Linux on x86, system modules are mapped to the same address for every process. As a result, the debugger can simply get the function addresses by resolving the addresses of the symbols in its own process. This convenience, however, is not existing on Linux on ARM. Getting the addresses in the debugged process memory space eventually boils down to determining the base address of the loaded kernel module in the remote process and extracting the function offsets inside the module binary on disk.

On Linux every process has its own info structure placed in the `/proc` folder. It includes a mapping of all currently loaded modules of the process. Searching for the name of the shared object that exports the send and receive functions allows to retrieve its base address in memory space of the process. It also has a reference to the location of the binary on the file system. Therefore, we use the *nm* utility application that gives us the offsets for function names in a binary file. Based on the fact that Linux maps the whole unchanged binary file to memory, the address of the send and receive functions in

process memory space can be simply calculated by adding the retrieved offsets from *nm* in the binary and the base address of the module in memory from the `/proc` folder.

6.4.1.2 Detecting Successive Calls to Function Pairs and Locating the Wrapper Functions

When the debugger stops at a breakpoint, it must verify that the call originates from one of the wrapper functions CaS or RaD. We implemented this by analysing the state of the remote program and excluding the case of internal encryption. Data is considered to be sent over the network, when two successive calls to the function pairs encrypt and send, and receive and decrypt are observed, which use the same data pointers. Therefore, the implementation saves the function arguments at breaks of encrypt and receive, and verify them when breakpoints on send and decrypt are hit. Since the syntax of all target functions is known, the approach of section 6.3.2 can be used to extract the function arguments from the registers and the stack. When a function uses a return value for one of the parameters, the debugger additionally breaks on the return address of encrypt and receive to extract the arguments there.

For the second type of validation, when trying to detect the CaQ, the implementation copies the whole buffer at every encryption call. We track the data per thread, together with a time stamp, and evaluate it at interrupts on send. We free the copied data either after it is sent or after a certain time without being sent. This is necessary, in case data is copied from an internal encryption and otherwise stay infinitely long in memory. We chose 15 seconds for the time out threshold.

Only when this verification is successful, the detector module has to intersect the call graphs in order to detect the address of the wrapper functions. The call graph at runtime can be obtained by reconstructing the stack frames. We implemented the reconstruction for applications that are compiled with the *-mapcs-frame*, described in table 6.1. Our implementation gets the address of the first stack frame from the frame pointer register. The pointer to the next frame pointer is stored at the retrieved address minus twelve. From there the next frame pointer is again in memory at the current frame pointer minus twelve. When all stack frames are reconstructed, the two call graphs can be intersected by searching for the first frame that appears in both call graphs and hence, has the same previous frame pointer.

6.4.2 The Collector Module

The collector module is responsible for extracting the plaintext data from the communication as soon as the detector provides the addresses for the wrapper functions. The extraction with the debugger simply uses the argument extraction from section 6.3.2.

As soon as it retrieved the pointers from the arguments of the wrapper functions, the data can be read, modified and so on. A faster implementation uses code injection and hooks the wrapper functions. Therefore, the module, a shared object, must be loaded into the remote process with the dynamic loader of Linux.

On a program start up Linux uses a dynamic link loader that maps all required shared objects to memory before the process is actually run. The loader itself is a shared library of the kernel and provides an API call *dlopen* that allows to load objects even at runtime. Executing this method in the remote process requires, similar to detecting send and receive, the base address of the loader object in process memory and the offset in the binary file on disk. Since the debugger cannot directly execute functions in the remote program, the call must be implemented by controlling the program counter and all registers of the remote process.

At first a backup copy of all register values must be created, in order to not crash the program after finishing. Afterwards, a string that contains the path and file name of the to-be-injected module is pushed to the stack and the function arguments, namely the stack pointer and the value 1, must be stored in the registers `R0` and `R1`. Then, the program counter is set to the address of the loader function and the link register which holds the return address is set to zero. This way, the debugger traps with a null pointer exception when the remote process tries to return to address zero after finishing loading. Eventually, the backup for the registers can be applied and the injection is completed.

The last step to fully integrate the loaded module into the remote process is to place the hooks at the wrapper functions and detour the execution to the injected implementation. Based on the introduction to hooking on ARM in section 6.3.4, the implementation substitutes the first instruction by a memory load to the program counter, which results into the branch.

6.5 Summary

With the rising demand for confidentiality and thus, encryption in consumer level and commercial software, security testing faces new challenges. Currently, existing testing tools only have poor or no support at all for encrypted network communications. That is precisely the reason why we proposed a generic solution to this issue by using the iDeFEND framework. The framework makes the encryption transparent and thereby, does not violate the security of end-to-end encryption. Since iDeFEND cannot be used on the ARM platform and nowadays many network applications are from the mobile sector and thus, use ARM processors, we transferred it to the ARM architecture. Additionally, we pointed out the limitations of the current framework and introduced improvements

to it. Our novel methods provide a more generic approach for security testing. We introduced a method that allows to inject test data into network applications that use message queues. Our solution detects and hooks the function that is responsible for encrypting and enqueuing packets.

Furthermore, we introduced a generic method to inspect the incoming unencrypted network data. Our method does not rely on the presence of a RaD wrapper function or even frame pointers.

With the extended iDeFEND framework we provide an interface to the encrypted channel of an application that allows already existing testing tools to work as intended, also on the ARM platform. Our improved framework decouples the testing of software from the actual encryption.

Chapter 7

Protection using Virtual Machine Introspection

Regular intrusion incidents emphasize that protection against network attacks remains a great challenge. Thus, Intrusion Detection Systems(IDSs) are indispensable in modern infrastructures. While state of the art Network Intrusion Detection and Analysis Systems focus on suspicious traffic, they lack an effective analysis of end-to-end encrypted communication activity. Contemporary Host-based IDSs complement traffic analysis by an additional fine-grained investigation on the host, yet, they are prone to local attacks. To approach both limitations, we combine the best of both worlds, by introducing an analysis system that operates from a level below the operating system. In this chapter, we present a bridge for IDSs which relies on virtualization extensions of modern Intel microarchitectures. This allows us to achieve an analysis of encrypted end-to-end communication channels and protect our framework at the same time. As such, we create an isolated analysis system that is resilient to local attacks. More precisely, we employ Virtual Machine Introspection (VMI) to effectively engage and hide hardware breakpoints from the guest to analyse decrypted traffic without disturbing the overall guest operation.

VMI grants us the ability to keep control over the target system, even if it is compromised. Further, in order to intercept the guest system, we engage and hide hardware breakpoints by leveraging the system's ability to intercept accesses to debug registers. To prevent potential exposures, we apply emulation techniques satisfying requests to the guest's debug registers. There is a plethora of previous work within the area of VMI providing stealthy guest system monitoring [73] [92] [44]. While implementing similar approaches, we employ VMI techniques as a vehicle for protecting our framework.

The evaluation is presented in detail in Chapter 9.

7.1 Attacks on Host-based Intrusion Detection Systems

A Host based Intrusion Detection System (HIDS) resides on the host they should protect. Hence, rootkits and attacks, which compromised the host machine are able to detect the presence of HIDSs, as their functionality and security mechanisms are commonly known. After compromising a host, malicious software can evade the IDS by disguising its activity to avoid pattern detection mechanisms of the system. If evasion is not an option, malicious code interferes with the IDS components to prevent it from detecting the malicious software in the first place or even altering the reporting mechanism to its advantage, e.g. the malware is not listed in the reports. [44]

Specifically the iDeFEND framework works like a HIDS and a compromised system can deactivate the security mechanisms. Further rootkits can also open additional attack surfaces by weakening the security of the host they managed to compromise. This gives follow-up attacks the possibility to infiltrate the machine as well. The security gained by iDeFEND relies on an uncompromised target machine, as it will only protect against attacks incoming from the encrypted data of the target application. If an attack gets passed the framework either through the specific application where iDeFEND is applied on or through the communication of another application on the system, the machine has to be seen as compromised. At this point the attacker has several options to counter the security solution by using the following weaknesses.

- (P1) Analysing the machine status can detect the presence of iDeFEND.
- (P2) iDeFEND can be deactivated, after detecting its presence.

The possibilities to detect and react to iDeFEND depend on the method used for data collection. As iDeFEND uses the functionality of a debugger, which is attached to the target application, the attacker can use debugger detection techniques to check for the presence of the framework. To detect the presence of a debugger for a given application (P1), the attacker may use several anti-debugging techniques [17]. As iDeFEND does not protect itself against these detection mechanisms, the attacker may then react to the presence of the debugger and can assume that a debugger is used for analysis. From this he can derive that iDeFEND is being applied as a security framework. On the other hand, the attacker may disable iDeFEND to allow further attacks to come through the channel, which should be observed by iDeFEND. Even if the presence of the security framework is not detected, the attacker can still assume the presence of iDeFEND. He may attack the host by revert changes, performed by the original iDeFEND framework, such as clearing breakpoints or restoring the programs integrity. In detail he can disable the breakpoints set by iDeFEND (P2). This can be performed by modifying or clearing the debug register content, which will move or remove the active breakpoint. Further the flags for used hardware breakpoints can be cleared on using the DR7 on the x86

architecture. As iDeFEND does not check the presence of its breakpoints periodically, the framework will not be able to react to the changes within the target process.

The second method used in iDeFEND to collect unencrypted data for analysis is using hooks on the Crypt and Send (CaS) and Receive and Decrypt (RaD) function. Using hooks or even software breakpoints to extract data from the target application will violate the code integrity. The attacker may have access to the original binaries and therefore identify hooks on functions in the application by analysing the byte signature of the binary (P1). When the presence of iDeFEND is detected, the attacker may replace the function hook with the original instructions and deactivate the framework this way (P2).

7.2 Hypervisor Extension

In this section we present a design choice, which allows us to introduce security mechanisms to protect the iDeFEND framework against attacks described in the previous section. As this is an extension to the iDeFEND framework, we are using the same structure. Therefore we will describe the *Detector*, *Collector* and *Monitor* modules with their upgrades to accomplish their original purpose.

7.2.1 Framework Design

We are using Xen as a virtualization environment. The Virtual Machines(VMs) managed by the hypervisor are also referred to as domains. The privileged domain, named *Dom0*, controls other, unprivileged guest domains and we make use of VMI techniques to interact with our target guest. The goal of VMI is to view the state of a VM from outside of the guest and take control over the execution flow of the guest [44]. This includes the ability to read and write arbitrary memory, register content, pause and resume the target machine. VMI also allows to register events for a given guest machine and handles them from without the VM. An arbitrary guest domain without privileges is called *DomU*. The application, whose encrypted network traffic we want to extract for analysis is located inside such a *DomU* machine. The modules of our security framework are set up in *Dom0*. We use a hardware virtual machine (HVM) to support Linux and Windows Operating Systems(OSs). We are setting up native debugging functionalities for our target VM over VMI. This way we are isolated from the target system and therefore secure from influence of a compromised *DomU*. Figure 7.1 depicts the framework used to detect intruders over encrypted channels using VMI. The figure shows the setup of the modules, as well as the basic architecture provided by Xen.

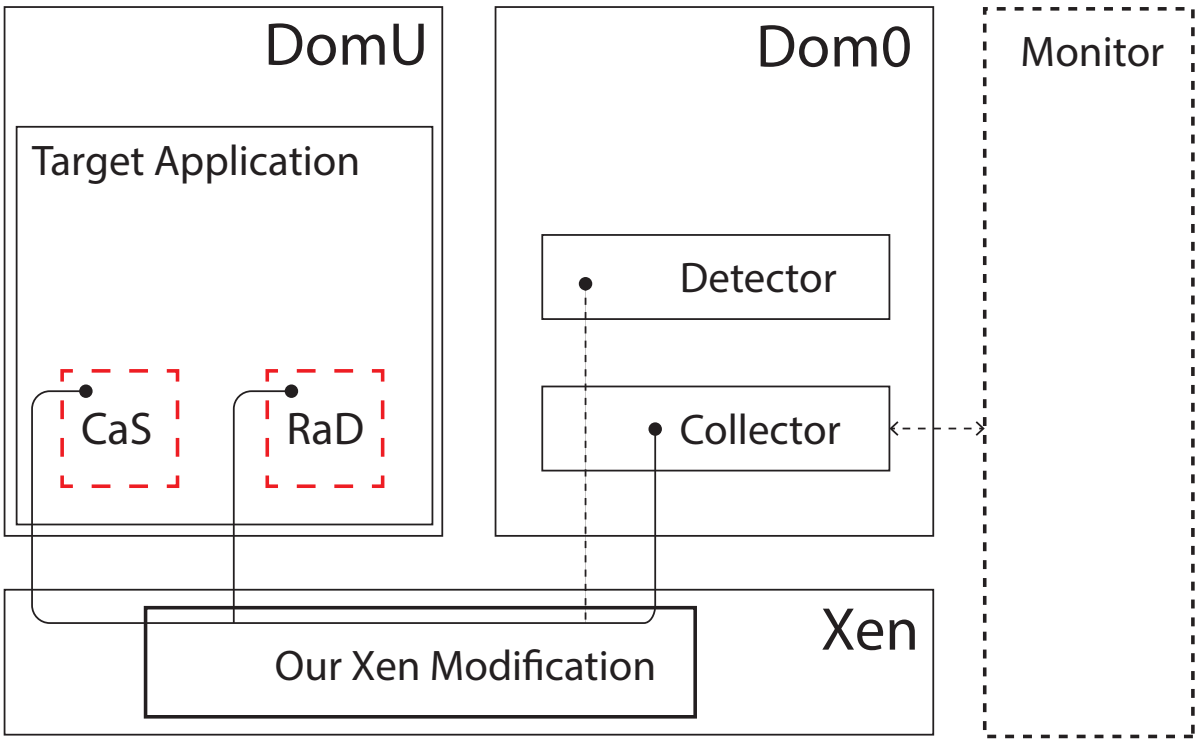


Figure 7.1: Hypervisor Extension

The *Detector* module in *Dom0* is again responsible for identifying the CaS and RaD functions of our target application and retrieving their Virtual Function Address (VFA). Detecting the relevant VFAs using the *Detector* can also be performed in a closed and safe environment, as the retrieved information has to be known once, to make use of the whole framework. As the *crypt* and *send* addresses are known from iDeFEND, we make use of these addresses. We install hardware breakpoints on the two functions using VMI. When the breakpoints get hit we can extract the stack from the target application to our *Dom0*. Here we can reconstruct both stacks, by applying guest OS meta information. After reconstructing the stacks for the hit breakpoints we can intersect the stack traces to find the CaS function and it's VFA. The important part about these functions is that they receive the transmitted data before any other function and have access to the plain text message as well. For CaS the parameters of the function will provide the plain text. On the other side the RaD function will receive encrypted network traffic and decrypt it. Identifying the VFAs of CaS and RaD of our target application allows us to set up the *Collector* correctly. The *Collector* module is described in its detail in the following section. The *Collector* makes use of the VFAs of the identified CaS and RaD functions. The hooking techniques described by iDeFEND are not needed for the hypervisor extension, as they were only used for security testing. We can extract plaintext data or modify the data as well just by making use of breakpoints at CaS

and RaD. We place protected hardware breakpoints at these locations. By protected we mean, that the OS is

- unable to read the debug register contents set by our framework.
- able to modify debug register values, without affecting the set breakpoints.

Using the placed breakpoints, we are able to extract the transmitted plaintext data from the application memory to the isolated *Dom0*, where our *Collector* is located. He passes the data on to the *Monitor*, which provides an interface for an IDS. This design choice allows us to apply the *Monitor* similarly as in the original iDeFEND. The monitor can be located within *Dom0* or on any uncompromised additional *DomU*. The *Monitor* allows to take actions on the result of the analysis of the provided plaintext data by the IDS. These actions include pausing the VM for further inspection or even remove the attack data inside the target application. There are three options for running the *Monitor* module. These possibilities include running the *Monitor* on *Dom0*, another guest VM or outsource it to another machine using the network. Running the *Monitor* on *Dom0* allows fast communication between the *Collector* and the *Monitor* as they use Inter-Process Communication (IPC) for data exchange. Setting up the *Monitor* on a separate *DomU* has the advantage of being really fast to set up, if the domain has been configured correctly once, as it can be ported to different hosts easily. Using the *Monitor* from a remote location may be slower than the previous two approaches, as the network presents a bottle-neck for the data exchange. However, using a remote location for the *Monitor* can be helpful for private networks, to use a shared IDS, without the need to setting it up on several machines.

7.2.2 Virtualized Collector

The *Collector* module is set up in *Dom0*, to be isolated from the target VM through virtualization techniques, as it runs on in a different domain then the target application of the framework. The constraint of isolation required for virtualized environments, defines that a guest machine, in our case any *DomU* may not influence the state or the execution of any other domain. The *Collector* will install protected hardware breakpoints inside the target application on a given *DomU* using VMI. The attacker can detect the virtualisation but not if the iDeFEND is in use, which partly solves (P1). This will be discussed in detail in Chapter 9. The protected hardware breakpoints we use, cannot be read from the guest OS, as we install a debug register access event, which allows us to change the content of the debug registers, before the actual read instruction is executed by the guest. Every time an access to the debug registers is required, the hypervisor will notify our framework, so we can modify the content or return fake values. When the breakpoint location gets executed, the collector gets notified through an event channel. At this point the target application is paused, as is the VM, and the *Collector* may use

VMI to extract the unencrypted network data from the memory of the process and pass it to the monitor for analysis. Step-by-step the *Collector* is performing the following operations.

1. Install memory event
2. Memory event allows to install hardware breakpoints, enable debug register access event
3. Debug register access event restores expected debug register content, before the OS access and enables the memory event again
4. Hardware breakpoint executed, allows us to extract plaintext data from the target application

First, we need to install the hardware breakpoints for the target application. As the target VM gets debugged in its running state, we have to make sure to set up the breakpoints for the correct process, as the target VM executes different tasks constantly. To modify the context of a process, we can use the hypercall API of Xen. However this function only allows to modify the context of the currently executing process. Therefore we install an event, which gets triggered if the memory page, where our breakpoint should reside in, gets executed. Such a memory event set up through VMI uses the page fault on execution to trap the event to the hypervisor. If the memory page is about to get executed we get notified in our *Collector* module residing in *Dom0* through a VMExit. At this point we may modify the debug registers of the target VM as we are paused in the context of the correct process. Therefore we set up the breakpoint addresses in the DR0-3 registers and enable the breakpoints by setting the flags accordingly in DR7. We delete the memory event after its first execution, as it is no longer needed. As we protect the content of the debug registers from the guest OS we also enable the debug register access flag on DR7. This way we force the guest OS to throw a debug interrupt when the debug registers are accessed, allowing us to react to the access. On a debug register access we want to restore their content as expected by the guest. The expected values for the guest can be read from the debug registers before installing our hardware breakpoints in our memory event. This way we inject arbitrary values to the debug registers, before the guest VM performs the read instruction, solving the first constraint for a protected breakpoint.

In order to guarantee the execution of our breakpoints inside the target application, we also set up the one-time memory execute event again, as it was used for the initial installation of the hardware breakpoints. When the memory event gets triggered again, we can set up our breakpoints again, before the actual breakpoint address is executed. This ensures that our framework cannot be deactivated from the guest OS. This solves (P2) described in Section 7.1 and grants the second property of breakpoint protection. As the memory event gets hit again we can extract the expected values of the guest OS for the debug registers and use them in further circles on, as well as set up our hardware

breakpoints as intended. When our hardware breakpoint finally gets hit in the execution of the target application, we can read the actual unencrypted plaintext from the process memory and pass it on to the *Monitor* module. As we want to stay undetected by a compromised VM we also have to allow the VM to attach its own debugger to the same application as we do. The location of the guest hardware breakpoints can be extracted at the point our memory event gets triggered. We install memory events for the guest breakpoints as well, allowing us to hold a maximum of four hardware breakpoints per memory page.

7.3 Implementation

In this section we are going to explain our proof of concept implementation of our framework using VMI.

7.3.1 Implementation on x86

We implemented our framework on the Virtual Machine Monitor (VMM) Xen [10] 4.6.1 as a virtualization environment and we make use of LibVMI (Release 0.12) [91] as a VMI framework. *Dom0* is hosted with Ubuntu 16.06 64-bit, while our guest VM is running Windows 7 Enterprise 64-bit with Service Pack 1 and one VCPU. The host machine uses an Intel Core i5-4670 CPU with 3.40GHz. LibVMI allows us to interact with the target guest application from outside of the VM. Using the framework we may set up callbacks for specific events inside the target OS and modify the state of the VM. We are using a small Xen modification, which allows the interrupt event of LibVMI to react to debug interrupts as well, which are thrown by hardware breakpoints. We use a *vmi_event_t* with the type *VMI_EVENT_INTERRUPT* to install a callback to react to debug interrupts inside the target guest. Using LibVMI we may also set up a *vmi_event_t* and configure it to represent our on-memory-page-execute as presented in the design section. We can do so by using the macro *SETUP_MEM_EVENT* and specify the address of our hardware breakpoint and use *VMI_MEMEVENT_PAGE* as granularity. In order for the event to get triggered on execute we set the conditional parameter to *VMI_MEMACCESS_X*. By performing these steps we can install a callback, which will then get triggered if the page where the breakpoint is located in gets executed. The event will get registered for the guest VM after we use the *vmi_register_event* call and pass our prepared memory event as a parameter. When the callback installed by our memory event gets triggered we need to take care of several things.

First we want to extract the expected debug register values from the guest to present the correct values, if the debug registers get accessed from within the VM. As we are in the

callback, which got triggered by an execution of our target process we can inspect the process' registers using the LibVMI function *vmi_get_vcpureg*. We save the content of the registers to an array for later use and proceed by setting up the hardware breakpoints we want to have installed using *vmi_set_vcpureg*. This function allows to change the value of a register of the current process context. We also change enable bit 13 of the DR7 register to get notified about debug register accesses in our callback for a debug interrupt. We only need the memory event once to set up the correct values for the debug register. This is why we delete the memory as a last step of the callback using *vmi_clear_event*. At this point we wait for the target VM to execute either the instruction with our breakpoint or to check the register values for suspicious content. In both cases the callback of our interrupt event will be executed and allow us to react accordingly. If the breakpoint gets hit during the process execution, we want to extract the relevant data from the target process. We can read the content of a virtual address of the target process using *vmi_read_va*. This way we can extract the plaintext data from the parameters or the return value and pass it on to the *Monitor* for analysis. After the data has been successfully extracted and passed on, we can wait for the response from the responsible IDS and continue the execution if there was no attack. If an attack got detected we may overwrite the data inside the target application with zeroes using *vmi_write_va*. To keep the breakpoint enabled and jump over the interrupt handler of the OS which follows after the hit hardware breakpoint, we move the breakpoint by the addresses instruction length. When this moved breakpoint gets triggered, we may install the original breakpoint again. If there is a debug register access about to happen, we want to write back the expected values to the VM and set up our memory event again, in order to guarantee the reestablishment of our hardware breakpoints before their instruction gets executed.

7.3.2 Implementation on ARM

We also implemented the framework on a Cubieboard3 [30], featuring an Allwinner Tech A20 dual-core CPU [2]. This processor contains two ARM Cortex-A7 [6] cores, implementing virtualization extensions and debug logic which provides means for setting hardware breakpoints. We used Ubuntu Utopic on a v3.16 Linux Kernel, modified by Thomas Leonhard[74] for the privileged domain and an Ubuntu Trusty on a v4.1 Stock Linux Kernel, with the `CONFIG_PID_IN_CONTEXTIDR` flag enabled for the unprivileged domain, on which the target application runs. In order to separate and isolate the iDeFEND system from the observation target, we used the same hypervisor as for x86, namely XEN. As the latest release of the hypervisor (v4.8) does not implement access to or virtualization of ARM debug logic, yet, a patch for XEN has been developed. This patch extends the XEN API by configuration functions for ARM debug registers and event handling mechanisms. In order to make use of the extended API, another patch

for the Virtual Machine Introspection library libVMI [117] has been implemented, which provides a more abstract interface to the new functionality.

Our implementation for the ARM architecture differs in certain points. While we also use libVMI in order to inspect the monitored guest, configure breakpoints and handle occurring events, the employed strategy and functions differ. In order to set HWBPs on ARM, register pairs in the privileged CP14 register group need to be configured. Access to this registers is only possible from within the hypervisor code in our setup. As the most recent XEN hypervisor version 4.8 doesn't export functions through which these registers can be set, we patched libXenControl and the hypervisor itself to add the required functionality. The applied patch was subsequently extended to libVMI, such that the new XEN functions can be employed, using the libVMI API. Thus, in order to set hardware breakpoints, we use our newly introduced functions `xc_cp14_{getreg, setreg}`. In contrary to the x86 solution, our hardware breakpoints hit on a match with one specific code address, committed for execution. As the latest version of the hypervisor didn't provide functionality to handle incoming debug exceptions on ARM, we extended the hypervisor patch to notify libVMI and make the current breakpoint context available. This required new handler functions to interpret incoming debug exceptions, which send a notification over the already existing XEN event channel mechanisms, in order to inform libVMI. With the introduced changes, we are able to configure hardware breakpoints and register callbacks in libVMI for debug events. Extraction of the observation data works analogous to the x86 approach.

7.4 Discussion

In this section we discuss the protection mechanism of our proposed framework from a technical perspective. The evaluation regarding the high level attacks against our framework and the experimental results are presented in Chapter 9.

By using the presented design we get several advantages in security when compared to the usage of a userspace debugger. First of all the OS of the guest does not get notified of the presence of a debugger, as we do not modify any process blocks or similar to state, that a debugger is present for the application. This implies that anti-debugging techniques, which rely on such additional process information to detect the presence of a debugger will not detect the presence of our framework. Also this property allows the VM to attach a debugger to the process, without getting notified that it is already being debugged.

The only way to detect the presence of our framework (P1) is to inspect the content of the debug registers of the application's context. This requires accessing the debug registers. In relation to the implementation of a debugger, using the Debug Architecture of

the hypervisor, some security considerations have to be given. The employed debug architecture provides a variety of ways on how to handle an occurring debug exception and protect the handling process. In this work, we developed a debugger component, residing in the XEN hypervisor. In this section we will refer to this component as Hypervisor Debug Component (HDC). We need to ensure the safe execution of debug exception handling and prevent unauthorized manipulation to means of debugging control. Thus the HDC needs to fully observe and control accesses to all registers, changing the state of the debug logic.

In order to put a hypervisor in full control of the hardware, the Architecture provides means of trapping important events, such as accesses to certain registers, occurring exceptions or interrupts, into the hypervisor. Whenever such an important event is encountered, the execution flow may be configured to be rerouted to hypervisor code. The hypervisor may then decide to handle the event itself and hide it from an overlying operating system, thus ultimately exercising full control over all overlying components. This general strategy can also be applied to retain control over the debug logic.

On the x86 architecture, we use bit 13 of the DR7 register to get notified by the guest VM that the next instruction will access one of the debug registers. We may alter the content of the debug registers to the VM's expectation and continue the execution of the VM. The guest will therefore read the altered debug registers and will not be able to actually read the values set by us. We reenable the hardware breakpoints using our memory event later on. This way we guarantee the execution (P2) of our breakpoints and keeping the protection of our breakpoints intact.

The ARMv7-A Architecture is different. By configuring the HDCR.{TDRA, TDOSA, TDA} bits of the Hyp Debug Configuration Register (HDCR), all valid *non-secure* accesses to relevant debug registers are trapped into the hypervisor. *Non-Secure* and *secure*, in this context, refer to a security state, introduced by the Security Extensions [6] of the ARMv7-A Architecture. While conventional userspace and operating system code is executed in *non-secure* state, the *secure* state is used for implementations of a Trusted Execution Environment (TEE). The HDCR configuration introduced above only traps *non-secure* accesses. Since the *secure* state is defined as trusted environment, we we do not expect any malicious activity there.

With the introduced configuration of HDCR and DR7 all access and manipulation attempts of required debug registers are trapped to the hypervisor. Thus, providing full control over the debug logic to our HDC.

7.5 Summary

In this chapter we discussed the problems and attack surface, which is present in modern HIDSs. In the virtualized environment we can make sure that the actual attack surface of the HIDSs is outsourced to the hypervisor level. In detail we showed how we ported the framework iDeFEND to the hypervisor level, granting us the possibility to inspect incoming encrypted network traffic for attacks. By using protected hardware breakpoints we can extract the necessary data as soon as the plaintext of the message is available inside the target application. After the data has been collected, the plaintext can be analysed for attacks by a generic IDS.

Chapter 8

Function Identification

Reverse engineering is a challenging task, requiring time and experience. Analysing given binary executables can be used to find code blocks with certain properties, possible bottlenecks and vulnerable spots of a system or to identify components and their relationships. In order to accomplish this, reverse engineers are forced to look into the assembly of the binaries, since most of the software in use is closed-source.

The main goal of reverse engineering is to determine the functionality of a given binary or to locate a specific functionality inside the executable. This allows modifications or to bypass certain functionality. Usually the gained information about the software behaviour and structure resides in the reverse engineers mind. Therefore, mastering reverse engineering takes a lot of time and is labour-intensive even for an experienced reverse engineer, it is a costly activity.

There exist tools for static as well as for dynamic analysis of executables to assist the reverse engineer. Static tools can be disassemblers, string searching tools, signature comparing utilities and others. IDA Pro [56] is an example for a well-known static analysis tool. It is a disassembler, which includes function name resolving of known API-calls and allows the analyst to view the assembly in a flow chart, which is representing the branches of the code.

Dynamic analysis is mainly performed using debuggers, allowing the reverse engineer to inspect the executed code at runtime, set breakpoints and look at the content of registers at a given execution. Furthermore binaries can be analysed during execution with the help of binary instrumentation, allowing the analyst to dynamically insert additional code into the execution flow of the application. Using dynamic analysis it is possible to extract variables, memory accesses, function calls and more during execution. Also with the already existing tools the reverse engineering process takes a considerable amount of time. Identifying certain functionality requires experience and patience.

We developed a framework to achieve progress in this problem. We use Dynamic Binary Instrumentation (DBI) as base to collect all executed functions, identified by their Virtual Function Address (VFA), inside the target application. The gathered information is saved to a database and is processed by using set operations on the data and represented using our framework. Using graphical visualisation techniques we display the data in a manner to the reverse engineer so that he can deduce the applications behaviour and structure, thus increasing the efficiency of his reverse engineering process.

Parts of this chapter were published [67].

8.1 Related Work

Different researchers try to develop tools to make the reverse engineering process more efficient and to speed it up. Quist et al. [96] presented a method using dynamic analysis to visually represent the execution flow of a program (focusing on malware), making the process of reverse engineering easier. They use the Ether hypervisor framework to monitor the execution of the target application and display the data to the reverse engineer in a processed manner highlighting the often executed portions. A visual reverse engineering system was already presented by Conti et al. [25] indicating that visual utilities speed up the work of analysts noticeably. Conti et al. present a way to analyse binary files, allowing the reverse engineer to gain insight into unfamiliar formats and structures.

An overview over the existing software visualisation tools is given by Diehl [32]. For example, Rigi [63] displays program structure and interaction. Rigi allows to analyse and document large software systems, when there the source code is available. The information about the system's evolution is visualised as directed graph to represent software modules. SeeSoft [37] can be used to visually represent the evolution of software source code. Eick et al. focused heavily on the software engineering part such as version control and static structure analysis, but also use profiling as a dynamic analysis to round up Seesoft. Reniers et al. [97] present their tool for software maintenance. The toolset is designed to keep track of software structure, metrics and code duplicates. These informations are represented visually. Trinius et al. [112] use visual analysis to quickly identify malware samples and classify the samples according to their behaviour as illustrated by the tool with treemaps and thread graphs. To do so they use a sandbox report and visualise it to the analyst. Using this approach it simplifies the analyst's work of classifying new malware samples into the families of already known malware.

Previous work on function identification focused on identifying cryptographic algorithms using dynamic analysis or static analysis of the binary. Wang et al. [116] try to identify cryptographic functionality using DBI. The assumption about the programs behaviour

that the message is processed after the decryption causes problems when it comes to identifying block ciphers, because they only get processed at the end of the message. Caballero et al. [22] extend the method introduced by Wang et al. and scan for repeatedly called functions. This approach is able to identify more algorithmic procedures, but also leads to false positives in loop-intensive applications. The proposed method of Caballero et al. was further developed by Gröbert et al. [51], who introduced a divide-and-conquer algorithm, analysing parts of the target application's source and merging them back together later on.

DBI is already widely used for performance analysis of all kinds like callgrind [83] for call graphs and cache performance analysis and Dr. Memory [19] to find memory leaks. In general there are different approaches to do DBI. One way is probe-based like Dyninst [21]. In this approach, so called trampolines are added in the executable and when they are executed they jump to the instrumentation instructions. This makes the code not transparent because the original instructions are overwritten with trampolines and are no more in the memory. A program can notice this and so prohibit the reverse engineering. But as an advantage trampolines can be executed fast without much overhead. The more flexible approach for analysis is the jit-based approach which is used by frameworks like DynamoRIO, Pintool and Valgrind [18] [27] [83]. It means that just before a block of instructions of the original application is executed, it is analysed and new instructions are dynamically inserted. This means that the main work of the frameworks is done during runtime of the analysed application.

Determining relevant functions and parts is the main task of the reverse engineer. Our framework is providing the reverse engineer additional analysis functionality to visualise and identify interesting functions. We provide reverse engineers a tool, which allows to identify functions of their interest and does not rely on the behaviour of the implemented algorithms inside the binary application. Instead, we use program state classifications to filter the specific Virtual Function Addresses (VFAs) with the interaction of the analyst.

8.2 Use Case Scenario

First, we describe a concrete use case to explain the whole process of function identification in detail. In the following sections we will refer to this example.

We have an analyst interested in the SSH connect function in the target of evaluation. To have a ground truth we choose the open source application PUTTY [107], which is a well-known application as SSH client. At the beginning, the analyst has no knowledge about the structure of the applications code. The target application is a black box system.

As first step, the analyst will get familiar with the Target Of Evaluation (TOE) from a user perspective, but no expert knowledge about security or reverse engineering is required at this point. Knowing the features, the analyst will start the TOE using our framework and will be asked to enter a label name for the next task he want to execute. A label describes the high level state of the application for the next time frame. Every time the analyst is changing his action, he also have to change the current label.

Some examples for a label could be *Open File*, *Enter Data* or *Connect*. The analyst decides how detailed the labels should be. Using labels like *Entering IP-Address of remote server* instead of *User input*, makes sense if the analyst want to distinguish between the different user inputs. To keep the example simple, we use only three labels in this chapter.

- *userinput* describes the time slot, the analyst enters some data.
- *connect* describes the time slot, the TOE is establishing a SSH connection.
- *init* describes the application start and the idle status of the TOE and all other actions, which are not part of the labels userinput and connect.

Since the labels are set by a human, the time slots are not exact and will have overlapping states. As an example, if we change the label from *idle* to *connect*, the *connect* label will contain data from the *idle* state. Therefore we will filter them out using our framework.

The filter is applied by the analyst and requires basic knowledge about computer science and mathematics. The analyst have to identify in which states of the TOE the searched functions are executed. In our example, the connect function is only executed during the active label *connect* and is never executed while the other labels were active. The filter rules are based on set theory. In our case, we need the set difference operation to exclude all the elements from the *connect* label, which are also part of the other labels. The results after filtering the elements will be just a small subset of the original data, and they will be presented to the analyst.

For the next and last step, the analyst requires reverse engineering know-how. The displayed data contains the VFA of the remaining functions and the count how often the function was executed. The count is an important value for the analyst, because the analyst can control in some cases how often the desired connect function should be called. If we are looking for the related functions of connect, they will probably be called only once during the whole execution time, if there is only one connection established.

The latest version of the User Interface (UI) is illustrated in Figure 8.1. On the left side, we see an enumeration of the elements (VFAs) in our database. Below of them, we find a list of the used modules, which can be used in the filter, which is illustrated at the bottom of the Screenshot. Other categories for filtering are shown on the right side. In

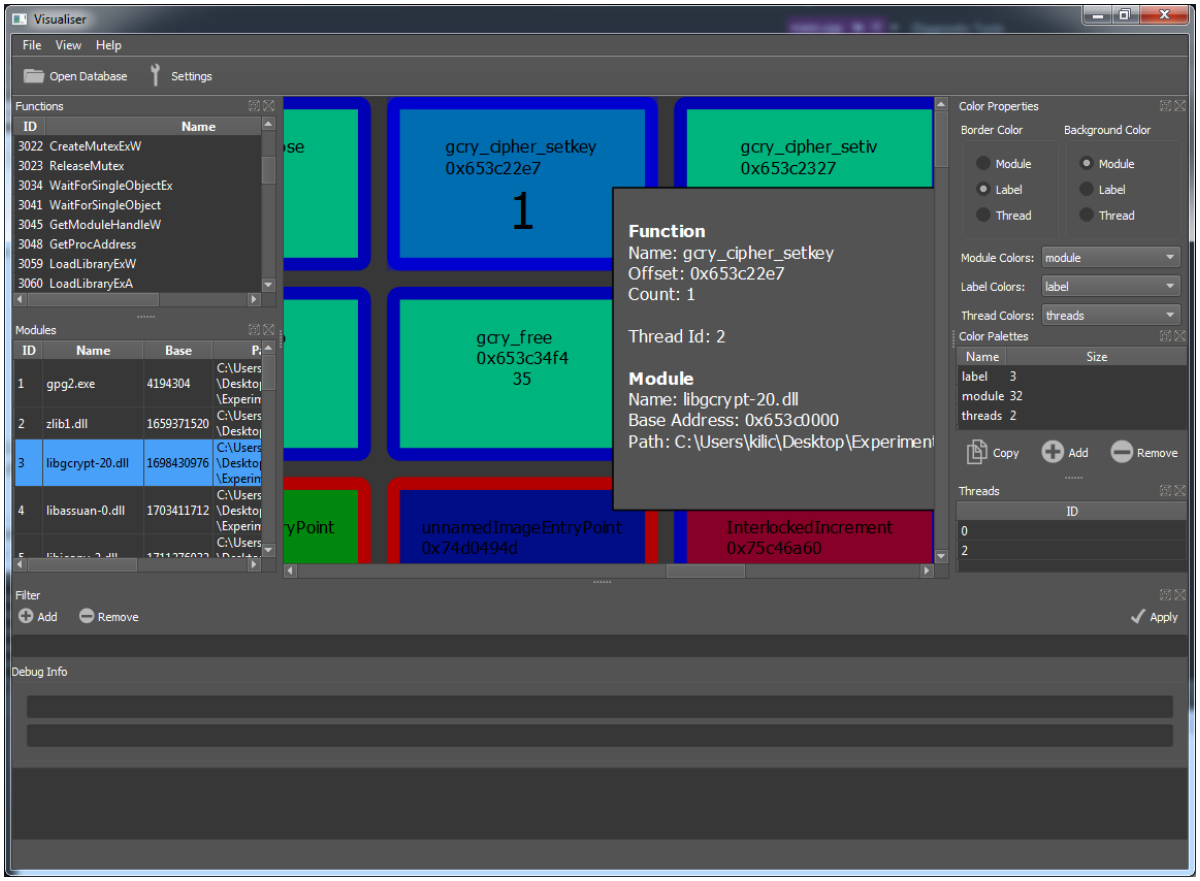


Figure 8.1: Screenshot of the Visualiser

the middle of the Figure, we can see the boxes. The border colour is representing the label and the background colour the module. The number is showing, how many times the function was executed. The box with the text is displayed, if the analyst in right clicking on a box. It contains all the meta data, which is associated with the VFA.

8.3 Application Design

Our Framework is divided into three components as illustrated in Figure 8.2. The *Extractor*-Module of our framework creates a real-time call trace of the target application. The interactive usage of the *Processor*-Module allows the analyst to filter the functions executed in a specific state of the program. The *Visualiser*-Module is responsible for visualising the information returned by the *Processor*-Module.

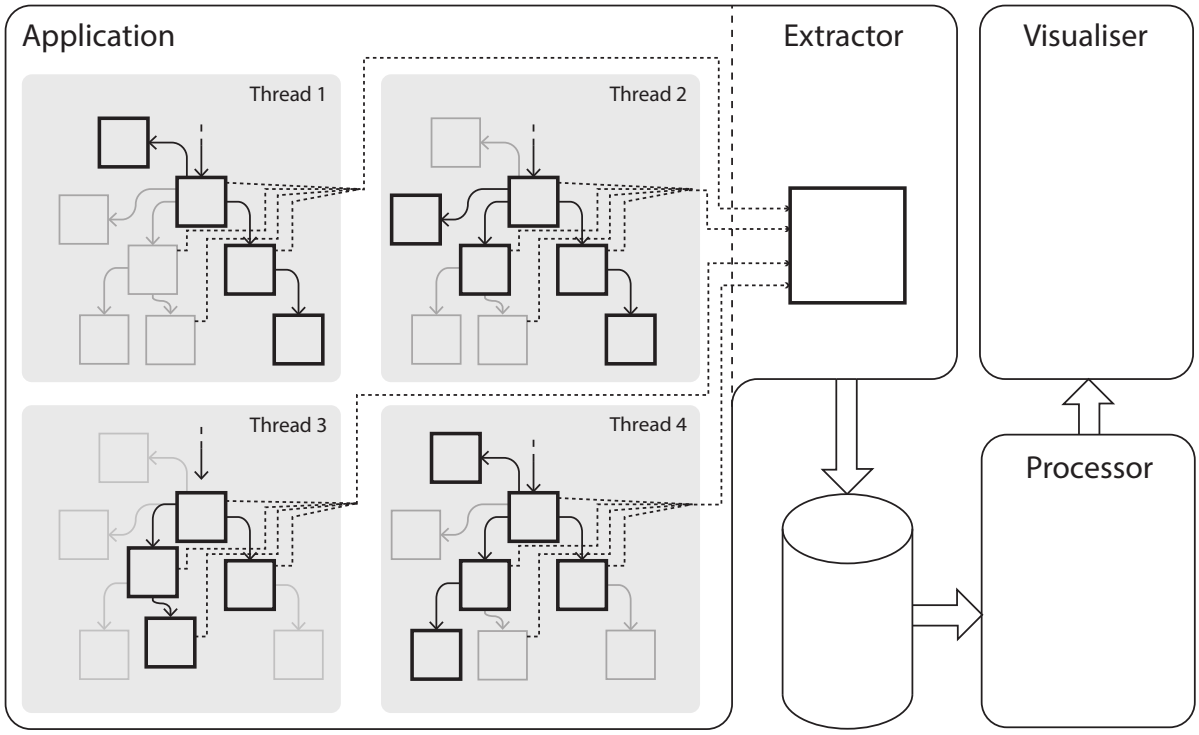


Figure 8.2: Framework Design

During the execution of the target application our *Extractor*-Module collects data on the loaded modules and the executed functions. By doing so, the *Extractor*-Module creates a call trace, which we want to enrich by a state property. The application will be in different states based on the actions the analyst triggers. A state in our case could be for example opening a file, establishing a connection or just idling. The analyst is able to set human understandable labels to mark the states that are interesting. By doing so the analyst marks executed functions of the current performed actions of the applications with a label. This label can also be interpreted as a state of the program, in which a set of functions are called to perform a certain action and lead into another state. We allow the analyst to set his labels with the help of our *Extractor*-Module, which adds the label information to every called function. This is performed using DBI. This labelling process is central to our approach, since we want to visualise functions executed in different program states later.

As further step, after collecting data of calls made during execution with the respected label, we allow to set different filters to the collected data. This functionality is implemented in our *Processor*-Module. Every element in our set is a VFA with additional meta data. The sets are defined during runtime based on three categories, which can be used for filter operations.

- *module* describes the name of the executable or library (data type: String)
- *threadid* describes the ID of the belonging thread as (data type: Integer)
- *label* describes the high level state information of the application defined by the analyst (data type: String)

The filter operations are defined as the set operation in mathematics.

- *Union* (X $\cup$ Y) merges the elements together between the both sets X and Y.
- *Intersection* (X $\cap$ Y) takes out all elements that are member of both sets X and Y.
- *Set difference* (X $\setminus$ Y) removes all elements from the set X, which are present in the set Y.

The syntax is defined as follows.

$$
\begin{array}{ll}
< \textit{category} > & ::= \textit{module} \mid \textit{threadid} \mid \textit{label} \\
< \textit{set operation} > & ::= \cup \mid \cap \mid \setminus \\
< \textit{value} > & ::= < \textit{String} > \mid < \textit{Integer} > \\
< \textit{set} > & ::= < \textit{category} > : < \textit{value} > \\
< \textit{set} > & ::= (< \textit{set} > < \textit{set operation} > < \textit{set} >)
\end{array}
$$

Example: ((*label*:connect $\setminus$ *label*:userinput) $\setminus$ *label*:init)

The *Visualiser*-Module is using the visual perception of human beings to display the results of the *Processor*-Module. It uses different optical elements like colours, shapes and sizes for fast visual recognition of important data. As one solution we use boxes for different functions, displaying the amount of calls for the function performed in the selected state filter. It adds colouring to the functions, representing *Labels*, *Modules* or *ThreadIDs* to give additional visual information to the reverse engineer allowing him to identify interesting code parts inside the executable faster.

8.4 Information Gathering

In this section we will first discuss different frameworks for DBI and then explain the decision for a framework we use in our tool set.

8.4.1 DynamoRio

The framework supports x86 (32-bit and 64-bit) and is available for Windows and Linux. It is developed as free software under the BSD license. To keep the application code transparent DynamoRIO follows three basic guidelines. The first states that we should keep as much as possible unchanged from the original application. So if you, for example, count the `mov` instructions in the program executed in the virtual environment it is very likely that the number is very near to the mov instructions execute in the original application. If it is necessary to change something, the application should not notice the change (second guideline). Finally the third guideline states that DynamoRIO does not make assumptions about the architecture and the operation system besides the minimum needed.

To generate basic blocks, DynamoRIO copies small parts of the executable into the code cache and applies small modifications. This is called copy-and-annotate. To optimize the execution, often sequentially executed basic blocks are combined and put into the trace cache which is separated by the normal basic block code cache. There the trace can be executed as one unit without executing management overhead, which decreases execution time. Also an indirect branch lookup is inlined and so it is possible to execute more basic blocks with indirect branches without doing a context switch back to the Manager. Another optimization DynamoRIO applies is the delay of interrupts. This is important because in contrast to Valgrind, for DynamoRIO it is not easy to determine the current machine context at every point. So if possible the interrupt is delayed to a point where the state of the application is accessible to DynamoRIO. For example if a timer signal is received and the execution is currently in the middle of a basic block in the code cache the application gets the timer interrupt later.

When it comes to the development of a tool for DynamoRIO (called clients), DynamoRIO provides the possibility to change the instructions of a clean C call before inserting it. Of course at this point the tool developer has to care himself about transparency and can deliberately destroy transparency. To avoid problems of shared libraries, DynamoRIO loads the library used in the instrumentation code separately.

8.4.2 Pin

Pin is a proprietary framework from Intel which can be used free of charge for non-commercial use. It supports x86 (32-bit and 64-bit), Itanium and ARM architecture. It focuses on an easy to use high level C/C++ API [27]. So Pin follows a call-based model which means you do not insert single instructions, only calls to C/C++ functions. Internally Pin works often like DynamoRIO and tries to improve certain points. In difference to DynamoRIO and Valgrind Pin automatically inlines code for performance optimization (mainly execution time) and takes care about register saving. Also it can be dynamically attached or detached to a program execution.

For optimization reasons Pin uses a Just-in-time-Compiler which directly compiles form the ISA code to the same ISA (for example x86 code to x86 code). During compilation, registers can be re-allocated. For example, if the instrumentation code, which should be included, needs registers also needed by the application, it can be avoided to save and restore registers by re-allocation of registers. This re-allocation must now be handled when going from one basic block (or trace) to another. Pin tries to do only the minimal reconciliation needed. To achieve this for every trace entry we remember the register bindings and consider them if we compile a new trace which targets this trace. No reconciliation is needed if we compile a trace which is a target of a single other trace. In this case we just use the binding of the traces targeting the trace currently compiled. Because of this technique, the executed instructions differ much more from the original instructions, compared to DynamoRIO.

8.4.3 Valgrind

Valgrind is an Open Source framework under the GPL licence and is available for many different ISAs like x86, ARM, PPC and MIPS. To combine the different ISAs, Valgrind uses an intermediate representation (IR). So first the code is translated to the IR and there the instrumentation code can be easily added platform independently. To execute a basic block in the IR the block again has to be translated back to the original ISA. This procedure is split into eight phases and is called disassemble-and-resynthesize.

Valgrind takes more basic blocks together to a superblock, which only has one entry but can have more exits, to reduce the management overhead. So Valgrind is one of the

most flexible DBI frameworks which of course also brings some drawbacks demonstrated in the next section. [82]

8.4.4 Instrumentation Tool Selection

For our framework we decided to use Intel's Pin [26] for three reasons. Pin can be attached and detached from the target application during execution, which is quite useful to allow the analyst to instrument only the part of the target application he chooses to.

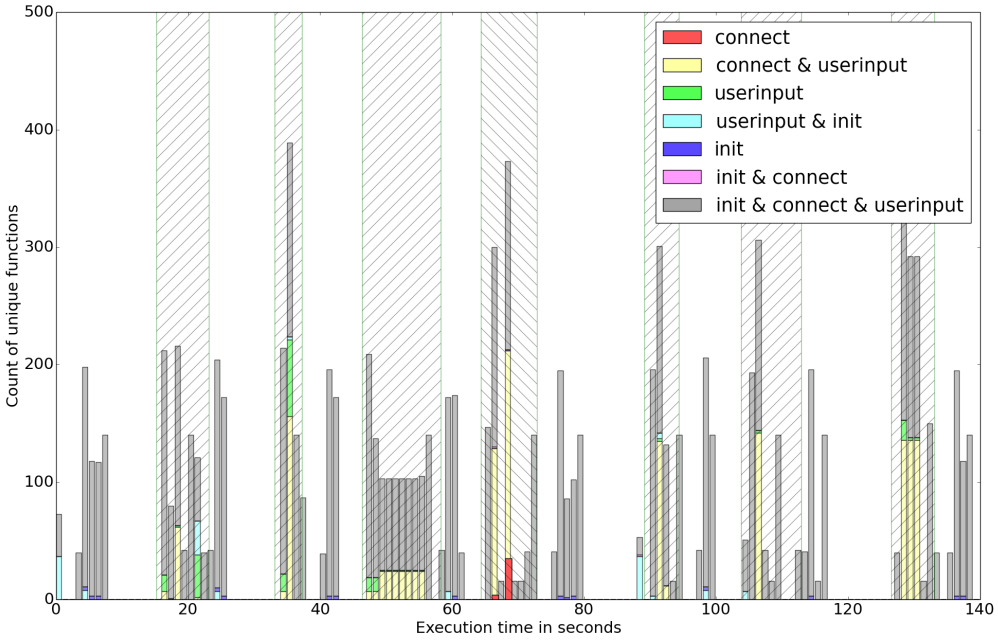
Compared to DynamoRIO, Pin is more stable. Memory intensive 32-bit applications can crash because DynamoRIO has a memory overhead which then could exceed the addressable memory [20] [18]. Also in difference to DynamoRIO, Pin chains basic blocks incrementally, which means that at the end of a basic block with an indirect jump, Pin adds new targets dynamically to the chain. Compared to DynamoRIO, which collects one trace and saves it, this approach is more flexible.

Valgrind is a much more comprehensive framework than Pin. Due to its great potential Valgrind loses out when it comes to performance. Regarding performance of an application with instrumentation, Pin is doing really well, compared to Valgrind and DynamoRIO, as stated in the Pin white paper [79].

8.5 Information Processing

In this section we present the *Processor*-Module of our framework. The gathered data contains the information about every executed function with a specific *Label*, *ThreadID* and *Module* matched to the respective VFA. The *Module* and *ThreadID* for each function is extracted from the application and is saved by the *Extractor*-Module. The *Label* information is set by the analyst using the *Extractor*-Module. A function with VFA x, e.g. the ssh_connect function, is part of the set label:y, e.g. connect, if and only if a call to the VFA occurred within the time frame, the specified label:y is active. Since the labelling is a time dependent property, the values in a set of a specific label are not limited to the specific functions the analyst is looking for. The target application is always executing some functions like updating the Graphical User Interface (GUI) of the application or something alike. Therefore the VFAs for a specific label may also have occurred in other defined labels. This has to be considered during processing.

Trying to identify a specific function in a binary, without having a valid signature of the desired function, is quite challenging. During the execution of the process we set our labels as high level program states. Figure 8.3 shows the execution of the PUTTY

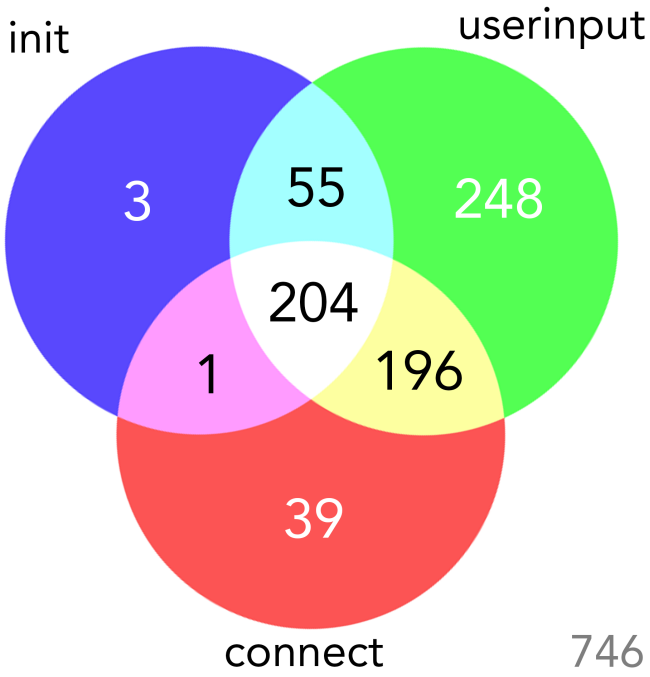


Amount of unique VFAs per second. Colours show the belonging of the function to labels or label intersections. // hachures show the time period where label *userinput* was set by the analyst, \\the label *connect*, *init* was set otherwise

Figure 8.3: Executed functions with label

application. The x-Axis shows the time in seconds, the y-Axis states the amount of unique function calls within a second span of time. The analyst is interested only on the functions that are part of the red bar. The time frame when label *connect* was specified is marked by falling hachures (\\) in the background, the label *userinput* with rising hachures (//) and label *init* was specified without hachures. The colours indicate the called functions belonging to a label or to an intersection of labels. We are interested in extracting unique calls of functions belonging only to the label *connect* coloured in red. The red colour displays the amount of functions uniquely called during *connect*. As we can see in Figure 8.3, while label *connect* is active, there are also function calls, which are also executed in the time frame of other labels. The goal of the *Processor*-Module is to process these interesting VFAs from the gathered data.

The *Processor*-Module provides the possibility to filter the collected functions by *ThreadID*, *Module* and the used *Label*, gathered during the execution. The main goal of this filtering functionality is to determine the specific VFAs that occur within specified circumstances (states). To do so, we allow the analyst to filter on the categories *Label*,



The numbers indicate the exclusive belonging of a VFA to a label or an intersection of labels.

Figure 8.4: Venn diagram of executed functions

Module and *ThreadID* with common operations on sets. These operations are union, intersection and set difference ($\cup$, $\cap$, $\setminus$).

To extract those functions from the gathered data we will use the set operations of our *Processor*-Module. As shown in Figure 8.4 after the extracting process we end up with 440 (=39+1+204+196) functions (red label set) called during the label *connect*. To reduce the amount of VFAs to the interesting application parts, the reverse engineer has the possibility to exclude the other two labels using our *Processor*-Module with its set difference operator, ending up with 39 VFAs uniquely called by the functionality he is interested in. Figure 8.4 shows the number of functions belonging to one or more labels. There are 39 functions, which exclusively belong to the label *connect*, while there are 196 functions called during *connect* as well as *userinput*. 204 functions have been called during all three labels. We can filter for the functions exclusively used in *connect* by using ((label:connect \ label:userinput) \ label:init). In this example we used three labels, but we are not limited in the number of labels.

After applying the analyst's filter, the *Processor*-Module calculates the call count for each function accordingly from the gathered call trace. The number of calls can help the analyst to understand the internal structure of the target application more quickly.

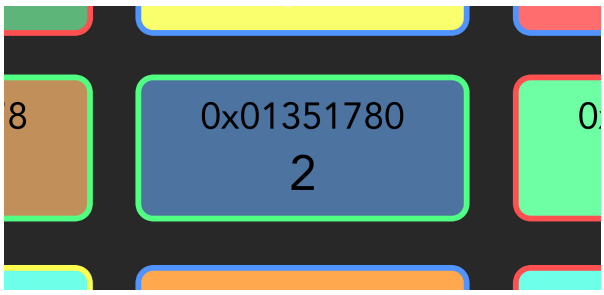


Figure 8.5: Graphical representation

While helper functions are getting called very frequently, wrapper functions are getting called less. Knowing this behaviour the analyst can focus first on the functions, which are rarely called. Therefore, we calculate the number of calls for the VFAs. Since the analyst knows how often he triggered his action during the program execution, the call count allows him to estimate if the VFA is relevant for him or not. E.g., if he triggered the connect functionality during the gathering process once he is most likely looking for a function also called once.

8.6 Graphical Representation

After the analyst specified the filter he wants to apply to the collected function call data in the *Processor*-Module, the results are presented to him by our *Visualiser*-Module. The main goal of the *Visualiser*-Module is to show the big data collection from the *Extractor*-Module in a manner that the analyst can quickly derive information about the program structure and behaviour by visualising the applied filter.

We propose a box view and a graph view to visualise the filter of the analyst. The result set of the VFAs of the applied filter is presented to the analyst by displaying boxes for every VFA as shown in Figure 8.5 by default. The VFA itself is not the first thing we want the analyst to notice. We want him to quickly identify functions with a low call count and highlight this property with text size.

Displaying the number of calls of a specific VFA gives the analyst a general idea about the structure of the program. VFAs with a smaller count call, especially when filtering for labels, tend to be the function the analyst is looking for. As a second focus after the call count of the individual VFAs we allow the analyst to bind function properties (label, module, thread) to colour the background of the box or the border. This gives the analyst multiple ways of analysing the gathered data and allows him to notice VFA properties visually. We use only two colour informations at the same time to not overload the user with visual effects. We provide a colour set as default, but the analyst may

Function property	Value	Visualisation
Function ID	385	-
Function Address	0x01351780	Text
Function Name	ssh2_setup_pty	-
Module:	putty.exe	Colour
Thread ID(s):	4	Colour
Label:	connect	Colour
Count:	2	Text and Size

Table 8.1: Function details and visual representation

change the colours individually. The method of visualisation of the properties is stated in Table 8.1.

E.g., after applying the filter ((label:connect \ label:userinput) \ label:init) for *connect* exclusive functions, we end up with 39 boxes in our *Visualiser*-Module. We bind the module property of the VFA to the background colour and the thread information to the border colour. This way we can quickly determine the main module responsible for the label we filtered for. We end up with 37 boxes of the module *putty.exe*, which represent the main VFAs for the connect functionality inside the PuTTY application.

Using this individual colouring option allows the analyst to familiarize with the VFAs and their properties on a visual basis, without having to remember properties in numerical form.

If the analyst is interested in a specific thread behaviour he may set the filter to that thread, bind the background colour of the boxes to the label property and the border colour to the module. The analyst can now identify the labels just by their colours, thus recognise the functionalities executed by the various threads.

Analysing modules for their purpose can also be achieved with the help of our *Visualiser*-Module. We just have to set the filter to a specific module. We bind the thread id to the background, to identify associated threads to a specific module just by the colour. Further the border colour may be set according to the label, giving us an idea in which program states the module is in use.

At the selection of a function box the analyst is provided with additional properties regarding the selected VFA. These properties include lists of labels and threads the VFA occurred, the module and the overall call count of the function (see Table 8.1.

The purpose of the graph view is to give a quick overview over the target application. It can also help the analyst when the result set of his filter is too large to narrow down his filter to the interesting functionality. In the node graph view the function VFAs are represented by nodes of the tree, the size of the nodes visually represents the call count

of the function if the analyst chooses to do so, allowing him to find VFAs with a lower call count more quickly. Further we allow the analyst to colour the nodes according to label, module or thread. We only display the call count inside the nodes and provide the additional properties (Table 8.1) of each function only on selection of the analyst.

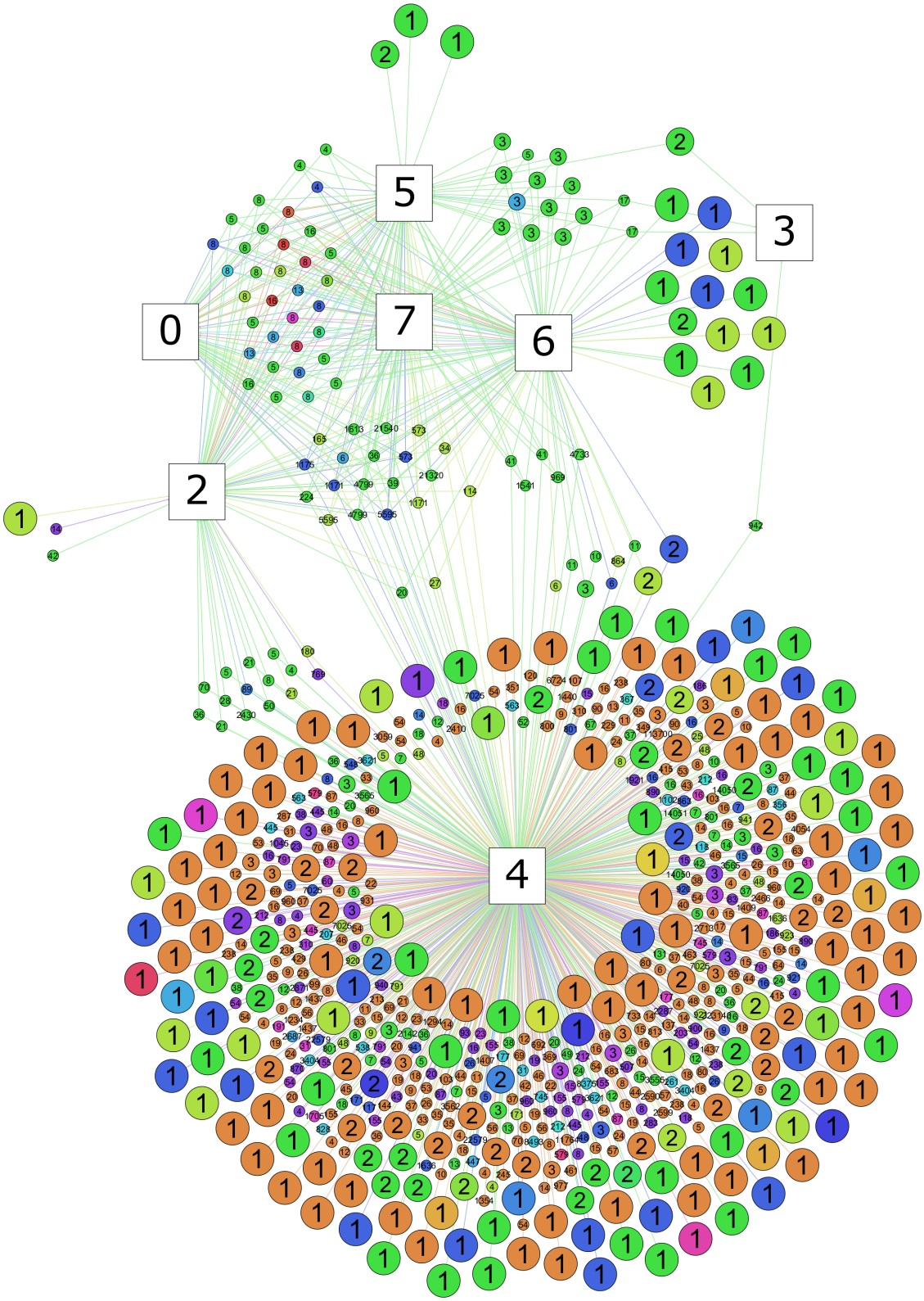
Figure 8.6 shows a representation of all the called VFAs (represented as nodes) related to their calling thread by edges. The rectangle nodes represent the different thread Id's. This allows the analyst to gain insight into VFAs used by one specific thread. For example the functions west of thread 2 were only called by thread 2. Thread 5 has its specific functions displayed north of the node and thread 6 to its east. On the other hand the analyst can quickly determine VFAs, which are used by more threads. In Figure 8.6 these would be the nodes between the threads 0, 5, 7, 6 and 2. Thread 0 and 7 only use functions shared with other threads. Such shared VFAs are most likely low-level functions, such as *strlen* or similar. Low-level and shared functions generally are called more frequently than specific functions, therefore we want to visualise the number of calls as a central focus of attention. The call count of VFAs is represented by the number in the nodes, as well as by the node size. The lower the call count the bigger is the node. This helps the analyst determine interesting functions as described in Section 8.5 in this graph more easily. Especially the VFAs with an amount of one or two calls are very promising to represent a certain functionality. From the graph the analyst can see how many different VFAs are used by the threads. Thread 4 calls the most amount of functions. The colour of the node represents the module, the function is belonging to. For example, the analyst can derive that thread 3 only uses one module, the dark green one. The orange module is only used by thread 4. Overall Figure 8.6 shows the analyst relations between threads, the functions and their modules, but also provides visual highlighting of VFAs with a low amount of calls. The used node graph is optimized with the algorithm ForceAtlas2 [60].

8.7 Implementation

In this section we describe how we implemented our framework on a machine with an Intel Core i7-4600U CPU 2.10GHz CPU and 8GB RAM. We used Windows 7 Professional with Service Pack 1 as Operating System (OS).

8.7.1 Dynamic Binary Instrumentation

In order to keep track of the called functions as well as logging additional information about the functions and modules we use Intel's Pin Framework (Pin 2.14 kit 71293 E) [26], because of its overall performance and stability described in Section 8.4.4



Nodes: VFAs; Rectangle nodes: ThreadIDs; Colours: Module; Number: Call count
Shows belonging of VFA to a module and threads.

Figure 8.6: Graphical representation with nodes stating the call count

The *Extractor*-Module is implemented as a Pintool, which has the purpose of saving the execution flow of the target application but also allow us to label certain states of execution.

The labelling process needs us to allow to interact with the Pintool while it is attached to the target process. To enable a simple communication with the Pintool we print the address of an allocated variable of the Pintool to a file before adding any kind of instrumentation to the target application. This address is then used by our labelling functionality. When the analyst sets a new label we simply open a handle to the target application process and use *WriteProcessMemory* to the passed address. From this point on the *Extractor*-Module will use the new label, with whom is written to the database. The overhead due to establishing an IPC would be even bigger. Thus we write the value into the memory ourselves.

After passing the address of the label variable we add two Instrumentations to the target application. The first, *IMG_AddInstrumentFunction* is called whenever an image is loaded by the target process. Therefore this instrumentation allows us to gather data about the loaded modules and save them to our database. The second, *RTN_AddInstrumentFunction* allows us to insert a function at routine granularity. This way we can enumerate the existing procedures, save the interesting data about these functions and add a further instrumentation using *RTN_InsertCall* to log every call of the function.

8.7.2 Buffering Data

Keeping track of all the functions called within the target application is a challenging task. Since we also want to instrument very fast and thread intensive applications, the amount of calls in the target application may quickly exceed the amount of calls we can save to our hard disk. If we would pause the process on every function call to write the desired information to our hard disk, we would end up with a very slow reacting binary. The amount of function calls may exceed the number of functions we can save to hard disk in a given time period. We want to buffer the gathered data in memory and write them to hard disk to improve performance. Since we gather data from different threads we have to make sure we are thread safe. The Boost Library [15] includes a single-producer-single-consumer lock-free queue. This means one thread is allowed to push to the queue and one to pop from it without worries about race conditions. We use a *boost::lockfree::spsc_queue* and keep the data in memory. We save objects representing function calls in our queue from multiple threads so we have to use a lock to take fall into the single-producer requirements for the queue. Furthermore we have to pause the target process if our queue is full. Otherwise we would miss some function calls in our execution flow. Writing to the database will be done by an additional thread in the target process. This allows us to push multiple function calls onto memory during program execution

without great performance issues. The additional thread will pop multiple strings of gathered data about the function calls from the queue, concatenate them and then store them to the database on hard disk. This way, we lower the performed hard disk accesses. We plotted the time related to the objects of called functions kept in memory for the worst-case-scenario under full CPU usage. As we can see in Figure 8.7 a usage of more than two hundred objects will not gather any benefit to the performance, since the time used per object settles down at about 0.02ms afterwards.

If the target application exits we pause the process, write the remaining data from the queue to our database and then quit the process.

8.7.3 Gathered Data

We want to save information about the modules loaded by the target process, like the name and path to the module for recognition purposes and the module base to calculate relative VFAs of functions. Further we are interested in properties of the called functions such as their name, the module they belong to, the VFA and the amount of calls of the function. During execution of the application we monitor the timestamp of a call, the VFA of the currently called function, the ID of the issuing thread and the label specified by the analyst.

All this data is saved to a single SQLite database (SQLite Version 3.8.10.1) [105]. We decided to use SQLite, because it is a lightweight solution. Furthermore SQL allows us to perform queries to evaluate the gathered data very efficiently. The scheme of this database is illustrated in Figure 8.8.

When a module is loaded by the process we save it in the table *modules*, storing its name, the path to the module and the module base address. Functions present in the executable are saved with eventual given name and VFA and a reference to a module they belong to. The call count is calculated after the extracting process, in the *Processor*-Module. The table *calltracking* is the main table where we store our call trace. For every function call we insert a new ID, with a timestamp, save the VFA of the called function and add the thread who called the function and reference the label as an integer variable.

8.7.4 Information processing

As a next step the analyst can use the *Processor*-Module, which can be used to perform common set operations to the gathered data. Since the data already resides in a SQLite database we use the SQL implementation of the set operations to reach our goal.

Union of two sets of filters is performed by concatenating two select statements, according to the desired filter together, together with the UNION operator. Intersections of sets

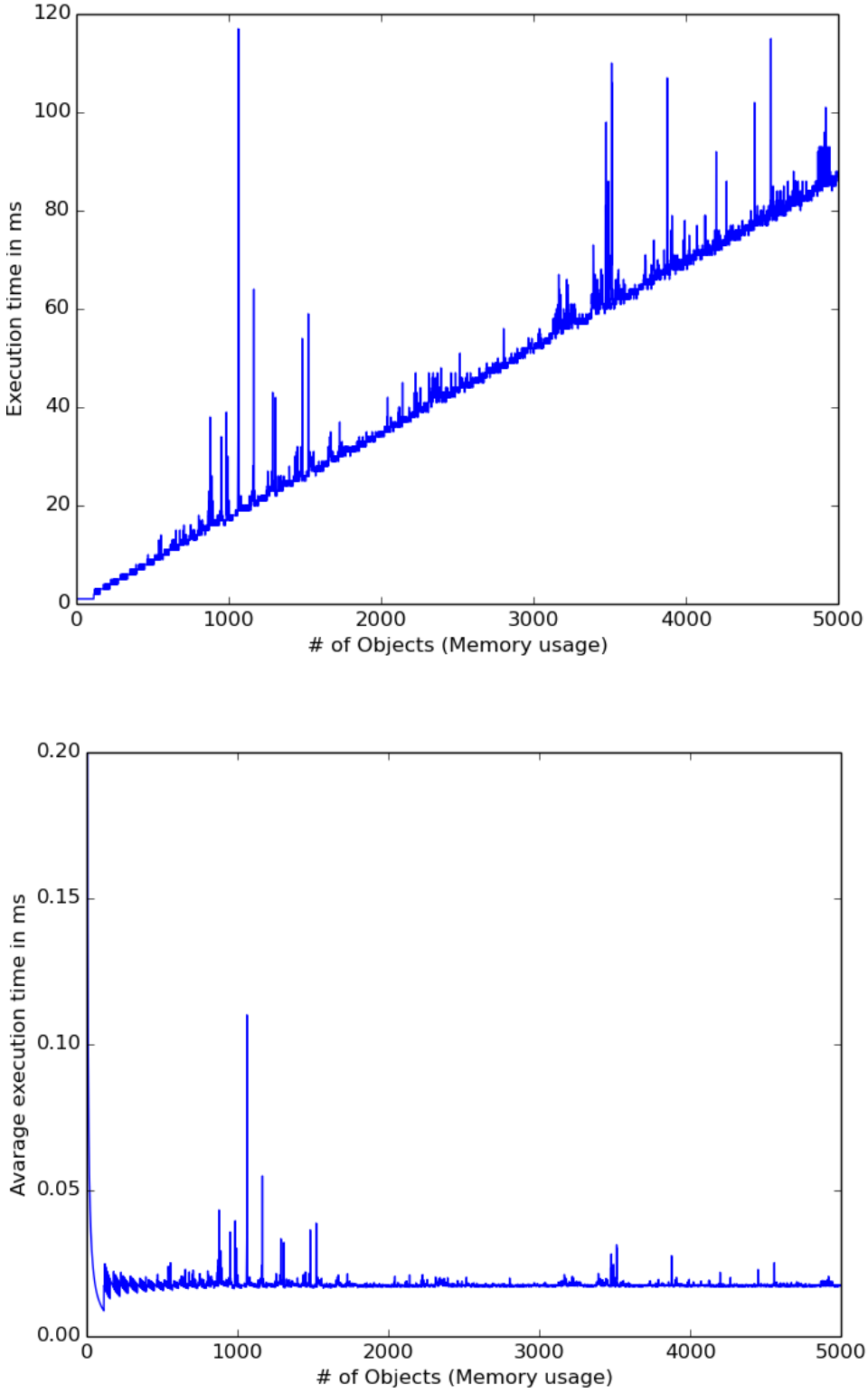


Figure 8.7: Time/memory plot for the worst-case-scenario

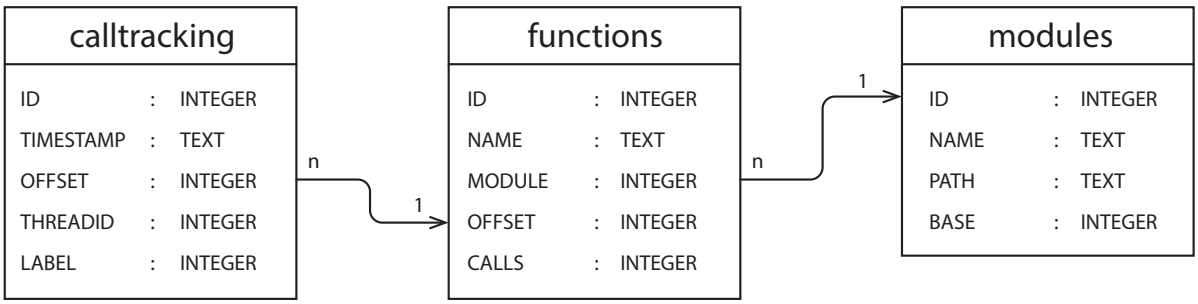


Figure 8.8: UML-Diagram of Database

are performed using the INTERSECT operator and the set difference is calculated by using EXCEPT.

After the gathering process we can make use of the *calltracking*-table of the database to apply filters to the label and the different threads. Since this table contains huge amount of data and increases very fast, a few hundred thousands of entries in a short time, we make filtering more effective by creating a temporary copy of the *calltracking*-table, restricting it to unique entries of *VFA*, *ThreadID* and *Label*. Our main output is the table *functions* where the VFA and the count resides. These are the main informations we want the analyst to catch immediately. We also apply module filters to the *functions*-table, since a join of the large *calltracking*-table with the functions table takes an unnecessary amount of time.

We also provide parentheses to be inserted into the filter to gain even more possibilities and lower possible mistakes in longer statements. E.g. the analyst can specify the following operation chaining together multiple set operations as he chooses.

$$((\text{label:key\_Enter} \setminus \text{label:key\_A}) \cap \text{threadid:7})) \cap \text{module:gpg.exe}$$

To provide the use of parentheses in filtering statements we use temporary tables. These are created for any statement between an opening and a closing bracket, parsing the input to a statement without any brackets, removing the innermost once at a time with regular expression search. This is also necessary because SQLite does not support the use of brackets with multiple select statements. Therefore standard SQLite statements could not reach the details we would like. The temporary tables are dropped before a new filter is applied.

8.7.5 Information visualisation

After applying the desired filter to the function calls, the analyst quickly wants to determine functions by their VFAs with special attention to functions with a low call count. A low amount of calls is indicating a higher probability of being the function, the analyst is looking for. This is due to the fact that the analyst is looking for a functionality, which is most likely implemented in a wrapper function. These are called less often than the according helper functions.

The *Visualiser*-Module of our application shows VFAs in boxes. These are implemented as custom widgets in Qt (Qt Version 5.4.1 32-bit), allowing us to bind the border colour or the background colour to the analyst specified preferences. On selection of a box we display the additional information about the VFA in a small pop-up window.

After the database selection we make sure to generate an appropriate amount of colours for the function properties and only show them if the analyst chooses to do so, as we do not want to overload the interface with information the analyst is not interested in.

The *Visualiser*-Module uses the box visualisation as the primary way of representing the functions according to the analysts filter settings. The node graph view as described in Section 8.6 still has to be implemented in future work.

8.8 Experiments

In this section we present some experiments performed with our framework. We show how we identified specific functions in open source software, explain the labels we set during the execution and provide the results of the performed tests on the framework's performance.

We used as a ground truth open source applications to validate the detected functions in the source file. We selected three applications using security related functions like cryptographic functions or hash functions. We have chosen the well-known applications Putty [107], GPG [50] and OpenSSL [86]. To verify our results, we compiled the test application with debug information in order to get function names for the found function VFAs.

We present the results of function identification in Putty, GPG and OpenSSL in Table 8.2. We set labels during execution for Putty trying to identify the functions relevant for establishing a SSH connection. We processed the data with our framework showing only functions that occurred during state *connect* and excluded all the other labels. In GPG we looked for functions responsible for creating an RSA key. OpenSSL is analysed for the relevant functions for the creation of the PEM file of a newly created key.

APP-Name	# executed	# filtered	# interesting functions	relevance	reduction
Putty	746	37	18 ssh functions	48.6%	95.04%
GPG	335	23	8 cipher functions 6 hash functions	34.8% 26.1%	93.13%
OpenSSL	735	29	24 cipher/hash functions	82.8%	96.05%

Table 8.2: Experiments

Step-by-step we are now going through the example looking for PuTTY's SSH connection functionality.
We started in label *init* and didn't interact with the application. We changed the label to *userinput* and typed the username, hit the ENTER key and typed the password for the connection we are going to establish. We used the *init* label between the actions. We change the label to *connect* and hit the ENTER key straight afterwards. After the connection is completed we change back to *init* and perform some more actions within the console like listing the files of the current directory and closing the connection using the *userinput* label. To identify the functions related to the connect functionality we used the following filter:

$$(((\text{label:connect} \setminus \text{label:init}) \setminus \text{label:userinput}) \cap \text{module:putty.exe})$$

The other applications were analysed in a similar manner in order to identify a main functionality.

In Table 8.2 the columns state the application name, the number of executed function and the number of filtered functions, which we found using an appropriate filter on the application. Further we number the interesting functions representing the functionality we try to identify.

The column *relevance* of Table 8.2 indicates how much of the filtered functions by our process also are interesting functions a reverse engineer might be looking for. The column *reduction* shows the percentage of the executed functions the analyst does not have to look into when using our framework filtering. With the help of our tool and using good labelling the analyst can reduce his work by reducing the number of functions to be analysed.

To understand the results better, a more detailed overview for the application Putty is given in Table 8.3. The time spend by the analyst took about 140 second to set the label and gather all the data. After that point the analyst set a filter, to reduce the data from 746 functions to 37. While the total number of all executed are 746, the functions related to the SSH functionality are only 236. From these functions are 74 executed while the label connect is active. Excluding all the SSH functions, which are executed

Description of executed functions	Total count	SSH related count
During the whole execution time	746	236
In the timeframe of connect	419	74
In the timeframe of connect Not executed in other timer frames	37	18
In the timeframe of connect Not executed in other time frames Total execution count is 1	22	12

Table 8.3: Statistics for Putty

while other labels are active, we get a total number of 18, which is listed in Table 8.4. These 18 functions are illustrated to the analyst together with a call count. Assuming the function we look for is executed only once, we have 12 functions left. Since the analyst does not know at that point if the functions are SSH related function, he has to check them together with the 10 unimportant functions, which are illustrated with a call count of 1 in Table 8.5.

8.9 Summary

We proposed our framework for interactive function identification decreasing the effort of reverse engineering. We have shown how we speed up the reverse engineering process by using three steps in the analysing process. We used the Pintool to get the data of all executed functions during the test. We used a time memory trade-off to speed up the logging functionally for real-time performance. We processed the data by supporting common set operations. A filter can be applied to labels, threads and module of a function and concatenated together using common set operations to restrict the desired functionality of the binary even further using our *Processor*-Module. We implemented a graphical representation for the output of the huge amount of data to quickly identify the results based on visual components. We used an iterative and interactive approach to set human readable labels using our *Extractor*-Module. We reduced the number of functions to be checked by the analyst.

Function name	Count
do_ssh2_transport	1
ssh2_channel_init	1
ssh2_chanopen_init	1
ssh2_chanreq_init	2
ssh2_msg_channel_response	2
ssh2_msg_channel_window_adjust	1
ssh2_pkt_defer	2
ssh2_pkt_defer_noqueue	2
ssh2_pkt_send_with_padding	1
ssh2_queue_chanreq_handler	2
ssh2_response_authconn	1
ssh2_send_ttymode	1
ssh2_setup_env	1
ssh2_setup_pty	2
ssh_agent_forwarding_permitted	1
ssh_pkt_defersend	1
ssh_setup_portfwd	1
ssh_tty_parse_specchar	1

Table 8.4: Relevant Functions for the Analyst

Function name	Count
conf_get_str_strs	56
bufchain_init	1
ctrlparse	1
alloc_channel_id	1
parse_ttymodes	1
term_get_ttymode	53
findrel234	56
newtree234	2
get_ttymode	53
strcmp	349
isspace	5
sscanf	1
vscan_fn	1
_ungetc_nolock	1
_filbuf	1
isleadbyte	1
_inc	13
_input_l	1
_whiteout	2

Table 8.5: Unimportant Functions for the Analyst

Chapter 9

Evaluation

In this chapter we will evaluate the iDeFEND framework. First, we will describe scenarios to show the problems in encrypted environment and how our approach solves it. Second, we will define the threats against our framework and evaluate our protection against dedicated attacks.

Parts of this chapter were published [65].

9.1 Scenarios

In this section we discuss some selected scenarios to describe the problems of encrypted network communication. Furthermore we also show how the problem can be solved with the framework described in this thesis.

Encryption in network traffic can be divided into two groups. First, a host-to-host based encryption, where all outgoing and incoming traffic from and to a host is encrypted and decrypted on OS protocol stack level. Second, the application-to-application based encryption, where applications use their own implementations of encrypting functions or are compiled with public encryption libraries.

In this work we considered only application-to-application based encryption scenarios. The reason is that for host-to-host based encryption, solutions are already present. If encryption and decryption are done on Network or Transport Layers of the OSI model, then OS features can be used to capture all incoming and outgoing messages after decryption and before encryption. For example, with IPsec/VPN we have encryption on Network Layer, so a message-capturing filter could be inserted between Network and Transport layers. An example of using Protocol Stack filters to capture decrypted messages in scenarios with host-to-host encryption is described by Abimbola et al. [3].

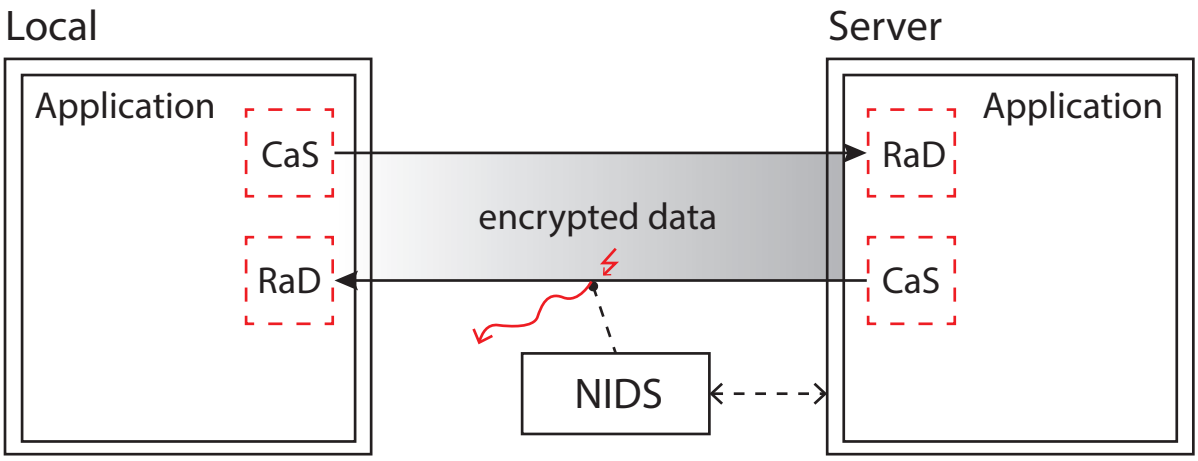


Figure 9.1: Scenario: Problem of Intrusion Detection Systems

All figures in this section are composed of two parts. On the left side we illustrate the local system, and the remote part on the right side. To refer to our framework, we use here Crypt and Send (CaS) and Receive and Decrypt (RaD) as the functions responsible for communication. CaS is sending data to RaD. Our solution is always part of one system, either local or remote.

9.1.1 Intrusion Detection

This scenario describes an application installed in the local network, which is communicating with remote applications located anywhere in the internet using an encrypted network communication channel. This application can act as a client or as a server depending on the use case. A Network based Intrusion Detection System (NIDS) is used to detect malicious data in the network traffic entering and leaving the local network. In this scenario, the NIDS can also be any Intrusion Detection System (IDS) or monitoring system, outside of the target operating system.

The problem we address in this scenario is that a NIDS cannot perform deep packet inspection of encrypted traffic (see Figure 9.1). Based on the encryption, accessing the plaintext data could also require knowledge about the used encryption algorithm and key. Considering signature based IDS, which inspects network traffic for specific byte sequence patterns, we need access to the plain-text data. One possible solution is based on terminating the encryption at a central point like a company proxy and send the data as plain-text to the receiver in the local network [48].

Another solution would be acting as Man In The Middle (MITM) and establishing a dedicated encryption channel from the MITM to the sender and the receiver. As an

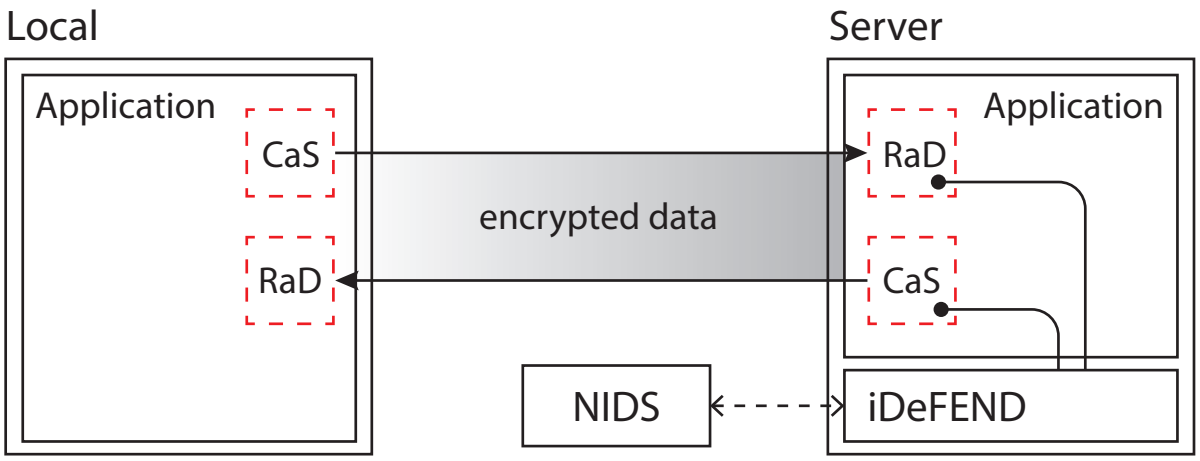


Figure 9.2: Scenario: Intrusion Detection with our Framework

example, this approach can be used to examine HTTPS traffic by installing company certificates as root Certificate Authority (CA) on the company workstations. The company can use a proxy server and terminate all HTTPS connections at a central point and generate new certificates and sign them. The web browser would accept this new certificates if the chain contains the trusted root CA.

But having propriety encryption would make this approach much more complicated and increase the effort by requiring reverse engineering. We propose a solution, which is illustrated in Figure 9.2, to inspect the plain-text data while keeping up the end-to-end encryption. Our solution accesses the data directly in the memory of the network application process right after it is decrypted or before it is encrypted. Additionally it does not require any knowledge about the encryption algorithm and key.

9.1.2 Testing remote applications

This scenario is considering a remote server application requiring an encrypted communication channel and a client, which can be installed in the controlled local network. Sources for both, the server and the client, are not available and the encryption algorithm and key are unknown.

The goal in this scenario is to ensure that the remote application is secure. As an example, fuzzing can be used to achieve this goal. Testing a remote server is easy if the communication is not encrypted. However, this is not the case with an encrypted communication channel. The server has to be able to decrypt the received message before it can start to process it. If the message is not encrypted correctly, the server fails to

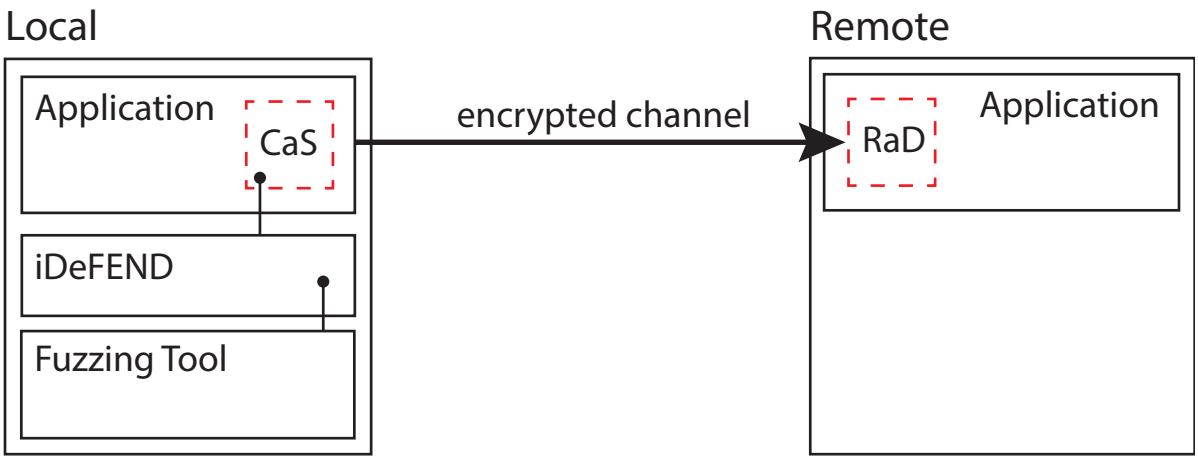


Figure 9.3: Scenario: Testing remote applications

decrypt it. Therefore, if a tester wants to cover server's logic beyond the decrypting function, it has to encrypt messages correctly before sending them to the server.

We propose a solution to test a remote server using encrypted communication. To encrypt messages correctly and send them to the server, the native client for the server can be used. As an example, we can intercept and modify the data the client is sending. While the local client tries to send a message, we stop the program right before it encrypts the message, replace the outgoing message right in the memory of the client with data from a security analyst or fuzzer-generated data. When the program resumes its execution, it will encrypt and send the data created by the modified data. We can also directly inject arbitrary data by directly calling the responsible function with our own parameters. This scenario is illustrated in Figure 9.3.

9.1.3 Testing local parts of a distributed system

In this scenario, we consider an application installed in the local network, which is under tester's control. The application is part of a distributed system and communicates with other parts of the system installed in networks, which are not under the tester's control. The communication channel is encrypted, the algorithm and key are unknown.

Vulnerable client applications, which are communicating over public networks, can also be attacked remotely.

In order to ensure that locally installed components do not have such vulnerabilities they have to be analysed. To test if the program is secure enough an analyst can use the same testing methods used by attackers to find vulnerabilities. We can distinguish

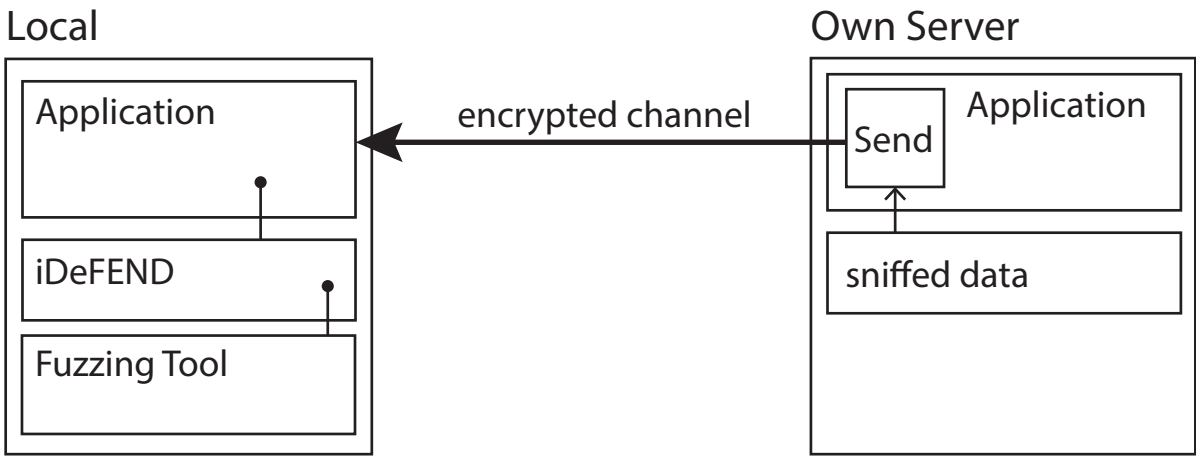


Figure 9.4: Scenario: Testing local applications

two cases for this scenario. In the first case the communication protocol is unknown. In this case the program can be tested only with random invalid data. The second case assumes there is some knowledge of the communication protocol, which can be used to make the testing process more efficient. The more precise the knowledge of the protocol is, the more efficient the testing will be.

The program under inspection can act as a server or as a client. Depending on the role the program plays when communicating over network slightly different techniques can be used to fuzz it. To test a client, we can enforce it to communicate with a fake server. After the local application decrypts the data, it will be replaced by the test data. We only need one valid network package, which we can replay many times. Such a message can be captured with a sniffer out of the communication between the application and a real server or client. The captured message could be sent repeatedly. The contents of the message do not matter, since it will be replaced anyway. The response of the local application will also be captured by our framework. Inspection of the data can show us, if the test succeeded or not. For example memory leak vulnerabilities like Format String Attacks(FSAs) or SQL Injections can be detected this way. Another way is to monitor if the application is crashing or not. This Scenario is illustrated in Figure 9.4.

9.1.4 Logging network data history

An environment for this scenario consists of a program installed in the local network communicating with remote programs using encrypted communication channel. The program may act as a server or a client.

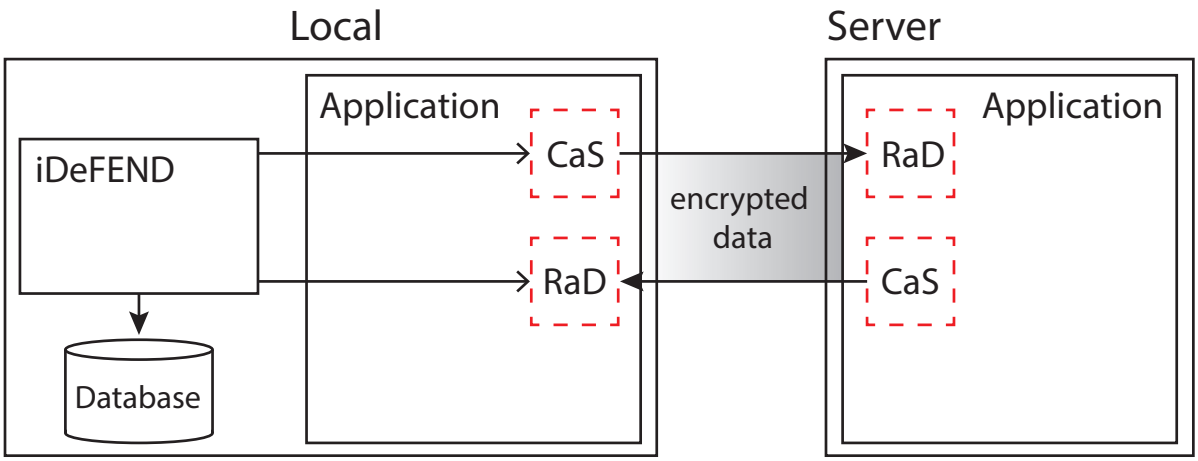


Figure 9.5: Scenario: Network data logging

Having a message exchanging history for the program which communicates over networks is useful for further analysis. Logging can also be achieved using a proxy as MITM, but this will break end-to-end encryption if the communication is encrypted. Extending an application with a logging feature is also not a trivial task if only the binaries are available.

Our framework is able to access a data in the memory of another process, it can be used to extend applications with a logging feature when solutions like sniffers and proxies do not help. According to the design of iDeFEND the logging can be done using the Monitor module. It can be considered as an additional advantage, that the original program is not changed. If we have certified applications, which are not allowed to be changed, we can basically enhance the functionality without modifying the binary. This is possible, because our framework will act like a debugger here and only inspect the application. Additionally, the newly added logging feature can be switched off/on even without restarting the application. This scenario is illustrated in Figure 9.5.

9.1.5 Behaviour change

An environment for this scenario consists of a program installed in the local network communicating with remote programs using encrypted communication channel. The program may act as a server or a client.

Behaviour of a program communicating with remote programs may partially or completely depend on data it receives. If we want to prevent the application from performing some actions without affecting its other functionalities, we can modify data right after it is received and before the application starts to process it.

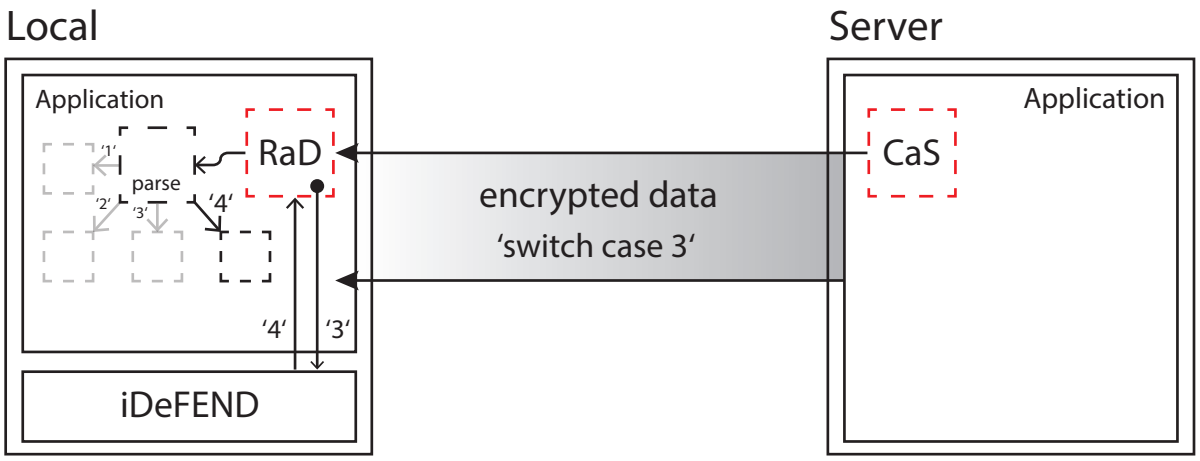


Figure 9.6: Scenario: Changing the behaviour of applications

Imagine the case, the application is polling on every start if a new update is available. If the remote server answers with a new update, the application will be updated by executing code on the server. A security analyst wants to test, if untrusted binary updates will be downloaded and executed by the target application, but currently no update is available by the remote host. In that case, the response of the server can be modified to fake that an update is available and can be downloaded.

With our framework we can access and modify data right after it is received and decrypted directly in the memory of the process under inspection. This scenario is illustrated in Figure 9.6.

9.1.6 Extending communication protocols

This scenario is about extending the communication protocol in a distributed system. One or more components of the system are installed in the local network.

Many applications developed long time ago using old technologies are still being used. But changing business requirements raises a need for software modifications. This is not problematic when sources for the program are available. If the program is no longer supported by original developers, and source code is lost, then the task of implementing the required changes is a challenging task.

Considering Industry 4.0, we have many legacy clients who are connected to the big infrastructure of the company. Since the system was not supposed to be available for intruders, no or less security mechanisms were implemented. As an example, consider

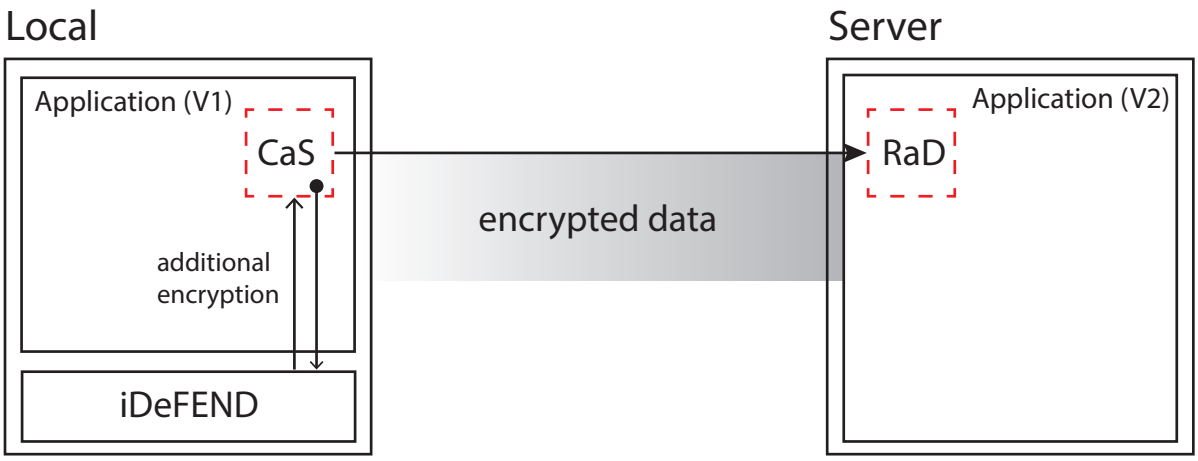


Figure 9.7: Scenario: Extending the protocol

devices communicating encrypted using legacy encryption algorithms, e.g. DES, which is known as unsecure in the current time.

Thanks to our framework we can stop the client before sending the required information over the network. We can extract the plaintext of the message, apply an additional secure encryption, e.g. AES-256, and replace the plaintext data with modified encrypted data. The target application will perform additional encryption using the DES algorithm and send it to the destination. Thus, we encapsulate the plaintext in our encryption, which is encapsulated by the original encryption algorithm.

Applying this solution to the infrastructure would require, that all parties have the same algorithm. Either, all legacy devices (v1) are enhanced with iDeFEND or additional new applications (v2) are part of the communication. This Scenario is illustrated in Figure 9.7.

9.2 Attacks against iDeFEND

In the following, we will analyse the attack surface of iDeFEND, define relevant assets and attack vectors and evaluate the hardening of the framework.

9.2.1 Assets and Attacker Motivation

As iDeFEND serves as an observation interface, which abstracts interaction with a given observation target for example an external IDS, the integrity and reliability of

the iDeFEND system and all information it is providing, is essential. By analysing the iDeFEND system according to the three classical information security goals (*Confidentiality, Integrity, Availability*) introduced by McCumber [80], we retrieve a set of security requirements iDeFEND should fulfil, which are depicted in Table 9.1.

Confidentiality	iDeFEND should not release information about the plaintext data, extracted from the observation target
Integrity	The integrity of iDeFENDs execution and the data it provides, has to be protected
Availability	The system and observation data flow should not be interrupted

Table 9.1: Security requirements for iDeFEND

A key property of the iDeFEND system is leaving the end-to-end encryption scheme of the monitored application intact. Protecting the confidentiality of the monitoring data therefore is an important security requirement. In order to ensure continuous monitoring of the observation target and provide reliable data to the external monitoring unit, the Integrity and Availability of iDeFEND are further important security requirements.

From the security requirements in Table 9.1 we can deduce the following key assets of the iDeFEND system:

1. Introspection Interface
2. iDeFEND Processes
3. Monitoring Data

The *Introspection Interface* represents the main connection between iDeFEND and the observation target. Manipulation of the introspection interface endangers Integrity and Availability of the system. The second key asset, the *iDeFEND Processes*, represent the core of the whole monitoring system. Manipulation or deactivation jeopardizes all three security requirements. The third key asset, *Monitoring Data* from the observation target, needs to be protected in order to enforce Availability and Confidentiality of iDeFEND.

While it is not necessarily known to an attacker, that iDeFEND is monitoring a target system, a set of general offensive goals can be defined. This goals include hiding malware and its communication as well as preventing countermeasures against malware operation. In the context of iDeFEND these goals can be reduced to the following.

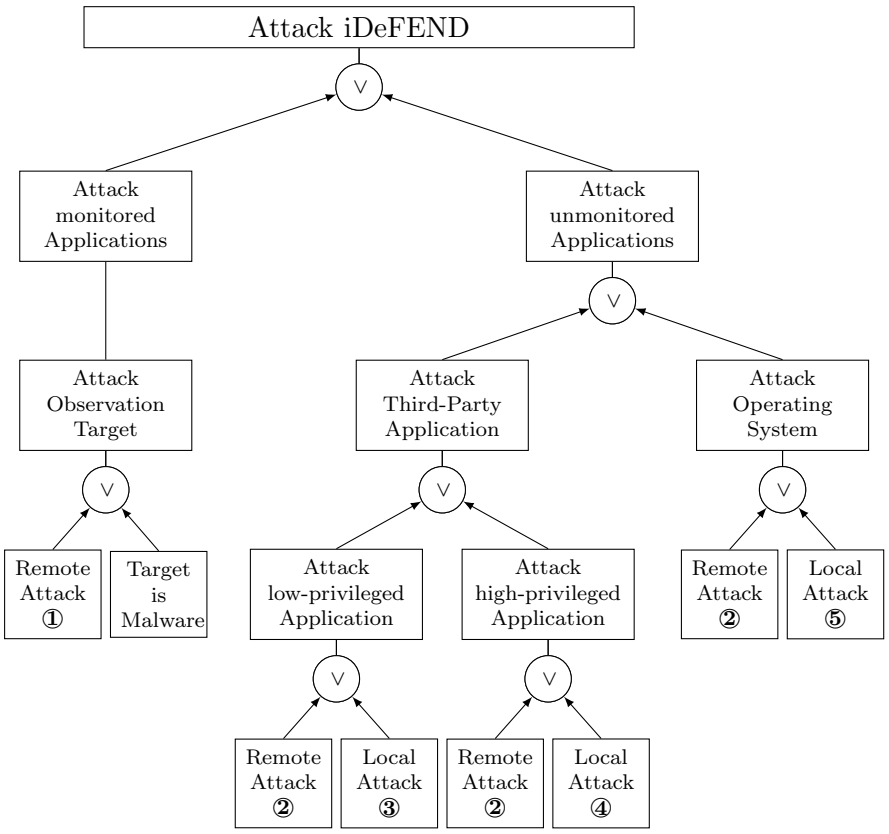


Figure 9.8: iDeFEND Attack Surface

- Detection of monitoring systems or components
- Disabling of monitoring systems or components
- Deception of monitoring systems or components

9.2.2 Attack Surface

Analysing the architecture of our framework, we can classify the attack surface of iDeFEND into different categories. This categorization is illustrated as an Attack Tree in Figure 9.8. Afterwards the introduced categories are summarized into attack vectors, covering the present attack surface.

Attacks on iDeFEND can be primarily categorized into attacks through monitored and unmonitored components. Such components are the observation target (*monitored*), the operating system and a third application running on the target host (*unmonitored*). While a third application may run with different privilege levels (*high* and *low*), every component can either be attacked remotely or locally. As iDeFEND is conceptually not

Nr	Type	Attack Vector
①	*remote*	Overtaken Observation Target
②	*remote*	Third network application
③	*local*	Third user-privileged application
④	*local*	Third high-privileged application
⑤	*local*	Operating System

Table 9.2: Attack Vectors of iDeFEND

designed to monitor malware itself, attacks through monitored malicious components are not discussed in this section. From the attack surface introduced in Figure 9.8, the general attack vectors shown in Table 9.2 can be deduced. These vectors resemble the markings from Figure 9.8 in enumeration and strategy.

Attacks on iDeFEND can be conducted through five general vectors in order to achieve the attack goals, defined in section 9.2.1. The first vector includes attacks, which are conducted through an overtaken observation target. The second vector consists in attacks through a third networking application, which is running on the same host as the observation target. Both of these vectors can be utilized remotely. Vectors three and four contain attacks through locally deployed malware, which is running on the same host as the observation target with differing execution privileges. Finally the fifth vector holds attacks from the target hosts operating system.

9.2.3 Attack Impact Analysis

In the following we will analyse the impact of attacks from the vectors defined in Table 9.2 on iDeFEND and compare the results. In the attack descriptions we will refer to the instance of the userspace implementation as *host* and for our extension as *host VM*. The attack analysis is mainly relevant for the operation of iDeFEND in a defensive scenario such as an interface for an external IDS.

9.2.3.1 Vector 1: Overtaken Observation Target

An attacker remotely overtook the observation target of iDeFEND through an exploitable zero day vulnerability.

Userspace Implementation The attacker is now able to detect iDeFEND monitoring, as the userspace implementation is not designed to hide its presence. Possible approaches

to implement the detection are behaviour analysis, timing attacks or requests to the operating systems debug interface. Depending on the deployed host architecture the attacker can interfere with the monitoring system in a limited manner, for example by manipulating breakpoints. This approach provides limited capabilities of temporarily disabling the monitoring process. However, a meaningful deception of iDeFEND is not possible through this vector, as all deviations from the expected behaviour of the observation target or monitoring outages, caused by breakpoint manipulations, are observed by the system.

Extension The management of the observation target is now conducted through VMI. While the detection of a virtualized environment is generally possible as shown by Thompson et al. [111], its presence offers limited guard against detection of the monitoring framework. Detection of monitoring through OS debug interface requests is no longer possible and detection through behaviour analysis is significantly harder. Due to the strong separation of iDeFEND and the observation target through virtualization, disabling or manipulation of the monitoring system is effectively impeded.

9.2.3.2 Vector 2: Third network application

An attacker remotely overtook a third network application running in userspace on the iDeFEND host or host VM respectively, through a buffer-overflow vulnerability.

Userspace Implementation Likewise *Vector 1*, the attacker is able to detect iDeFEND monitoring of the observation target running on the system. However, disabling or deception of the monitoring process are easily prevented by executing the iDeFEND processes with higher privileges.

Extension Likewise *Vector 1*, the attacker's ability to detect the monitoring process is limited while resistance against disabling and deception of the monitoring system is improved by separation through virtualization instead of separation through different execution privileges.

9.2.3.3 Vector 3: Third user-privileged application

An attacker managed to execute a malicious script on the iDeFEND host machine or host VM respectively, running with user privileges.

Userspace Implementation Similar to *Vector 2* the attacker may detect iDeFEND monitoring but is not able to disable or deceive with the process in any way due to lower execution privileges.

Extension Similar to *Vector 2* the detection of monitoring is limited and the attacker is not able to disable or deceive the monitoring process in any way due to separation through virtualization.

9.2.3.4 Vector 4: Third high-privileged application

An attacker connected a malicious USB stick to the iDeFEND host system or host VM respectively, which is subsequently infected with malware, running with elevated privileges.

Userspace Implementation Due to the high-privileged malware the attacker is not only able to detect iDeFEND, as described in the previous attacks, but also to disable the monitoring system completely. This may be achieved by interference through the debug interface of the operating system or simply stopping or removing the monitoring system. A sophisticated attacker may manipulate the control flow of either or both the observation target and iDeFEND in order to deceive the monitoring process, hiding the infection and its communication.

Extension While the detection potential stays the same as in *Vector 1*, disabling the monitoring system is completely prevented by the separation through virtualization. Deception of iDeFEND is possible in a limited manner, as a process with elevated privileges may potentially tamper with the monitored process. However, such attempts will be observed by iDeFEND in any case.

9.2.3.5 Vector 5: Operating System

An attacker managed to install and run a malicious kernel module on the iDeFEND host system or host VM respectively.

Userspace Implementation With full control over the operating system, the attacker is able to detect, disable and deceive the monitoring process as described in *Vector 4*.

Extension With full control over the operating system, the attacker still only has limited capabilities to detect, if the observation target is being monitored and is not able to disable the monitoring process in any way. He may interfere with monitoring by tampering with the observation target process in order to try and deceive iDeFEND, but like in *Vector 4* this attempts will be detected.

9.2.4 Summary

As demonstrated in the above analysis, the iDeFEND extension provides a considerable gain in protection against the attacks from the vectors, minimizing the overall attack surface. A summarized comparison between the defensive properties of the userspace implementation and our extension is provided in Table 9.3, which displays the respective defence attributes of the userspace iDeFEND implementation and our extension using VMI. Attributes are rated with **- (No protection)**, if no preventive measures against the respective attack are in effect. Attributes rated with **+ (Limited protection)** offer partial guard against the respective attack, whose special constraints are defined in the detailed analysis. Finally, attributes rated with **++ (Full protection)** render the respective attack infeasible, thoroughly protecting the system. In direct comparison, Table 9.3 shows, that the userspace implementation of iDeFEND lacks defensive capabilities regarding attacks with the goal of detecting a monitoring framework and attacks which are performed with higher than user privileges (4, 5). These shortcomings are patched by our implemented extension. The extension offers a limited protection against detection attacks, as some sort of additional behaviour analysis needs to be conducted. Furthermore, the extension protects iDeFEND against attacks from higher privileged processes or even the operating system itself by strong separation through virtualization. With the extension in place iDeFEND is fully shielded against disabling and significantly hardened against other forms of manipulation. Summing up, the extension enables iDeFEND to protect its key assets and fully comply with the security requirements, introduced in Subsection 9.2.1, in case of substantial infections of the observation targets host system, while impeding monitoring detection.

Userspace Implementation

Vec / Goal	Detection	Disabling	Deception
Vector 1	−	+	++
Vector 2	−	++	++
Vector 3	−	++	++
Vector 4	−	−	−
Vector 5	−	−	−

Extension using VMI

Vec / Goal	Detection	Disabling	Deception
Vector 1	+	++	++
Vector 2	+	++	++
Vector 3	+	++	++
Vector 4	+	++	+
Vector 5	+	++	+

− : No protection + : Limited protection ++ : Full protection

Table 9.3: iDeFEND Defence Attribute Comparison

9.3 Validation

We have evaluated our iDeFEND framework for five popular and well-known applications. Since the framework was designed to work on proprietary software, we only chose applications for evaluation that do not use standard libraries for implementing the encrypted communication. When a standard library is used, functions can be identified and hooked by simply looking at the export tables. Beside the required criterion of encryption, we wanted to have at least one messenger, one file transfer and one secure shell application. These types implement different network protocols which handle text messages, binary files and customized commands. Furthermore, we wanted to have at least one test application that is single-threaded, multi-threaded, uses the console for user interaction and implements an own GUI. In order to have ground-truth information of the wrapper functions, we used the following open source applications.

- telegram-cli, version 1.4.1
- uTox, version 0.7.1
- PLINK, version v0.67

- PSFTP, version 0.67
- PSCP, version 0.67

Table 9.4 gives a detailed overview of the selected applications. The second column states the type of the application. The third column shows the type of data that is primarily transferred by the protocol. The last two columns indicate whether the application implements a GUI or is multi or single threaded, respectively.

We evaluated the framework in three steps. In a first step, we compiled the applications from source. We used the *gcc* compiler with the *-mapcs-frame* flag. Afterwards, we attached the *gdb* debugger to the applications and looked for the used send and receive methods. This was realized by inserting breakpoints on the operating system network socket functions and waiting for them to be called.

In a second step, we reverse engineered the cryptographic functions and used the *gdb* debugger to manually detect the wrapper functions by intersecting the call graphs at crypt and send. We verified the result by analysing the source code with the help of the software *doxygen* [54]. Doxygen automatically generates a documentation for source code and improves the readability and understandability of function dependencies by providing visualized call graphs.

In a third step, we configured the framework with the function names of *send*, *recv*, *encrypt* and *decrypt*. Afterwards we used the framework to detect the wrapper functions and tried to inspect, intersect and inject data to the communication channel.

Table 9.5 summarizes the results of our evaluation for our userspace implementation. The first column contains the name of the applications. The columns send and receive state the system library functions the application used to communicate over the network. The column Wrapper-Type states whether a CaS or EnCrypt & EnQueue (CaQ) function is implemented. The following three columns illustrate whether plaintext data inspection, interception or even injection to the encrypted communication was possible by using our framework. The last column indicates whether code injection and hooking of the wrapper function was working.

Name	Type	Data Category	User Interface	Threading
telegram-cli	Messenger	Text	Console	Multi
uTox	Messenger	Text	GUI	Multi
PLINK	Secure Shell	Commands	Console	Single
PSFTP	File Transfer	Files	Console	Single
PSCP	File Transfer	Files	Console	Single

Table 9.4: Description of the open source test applications for iDeFEND

Name	Send	Receive	Wrapper	Inspect	Intercept	Data Inject	Module Inject
telegram-cli	Write	Read	CaQ	✓	✓	✓	✓
uTox	SendTo	RecvFrom	CaS	✓	✓	✓	✓
PLINK	Send	Recv	CaS	✓	✓	✓	✓
PSFTP	Send	Recv	CaS	✓	✓	✓	✓
PSCP	Send	Recv	CaS	✓	✓	✓	✓

Table 9.5: Results of the Evaluation for Userspace iDeFEND

Name	Send	Receive	Wrapper	Inspect	Intercept
telegram-cli	Write	Read	CaQ	✓	✓
uTox	SendTo	RecvFrom	CaS	✓	✓
PLINK	Send	Recv	CaS	✓	✓
PSFTP	Send	Recv	CaS	✓	✓
PSCP	Send	Recv	CaS	✓	✓

Table 9.6: Results of the Evaluation for iDeFEND with the Virtualization Extension

Briefly summarized, we were able to inspect, intercept and inject data for all five applications. Except for Telegram, all applications implement the CaS function. Telegram implements a message queue and therefore, uses the CaQ. We were also able to use code injection and hooking of the wrapper functions on all five applications.

Using the virtualization extension, the functionality is limited to inspect and intercept, which are enough for the use case of adding a bridge for IDSs. Table 9.6 summarizes the results for our virtualization extension. Module Injection is not possible, because modules will be part of the target system and therefore attackable. The responsible functionality of arbitrary data injection is also part of the injected module.

The focus of the extension is security and protection against intruders. In order to evaluate the protection against detection, we attach our debugger to a Virtual Machine (VM) and ran application protection tools. These tools have the option of protecting binaries from debugging by recompiling the binary itself. The tools in use are Themida (Version 2.4.1.0) [109] and VMProtect (Version 3.0) [104]. Themida claims to implement anti-debugging techniques to detect any kind of debugger, protects against memory dumping techniques and even more. VMProtect also uses a virtual CPU to execute the code, which has to be protected. This way VMProtect protects against further virtualization and code mutation. The toolkit also detects virtualization environments and the presence of debuggers. We install the protection tools on the guest VM and recompile the binary we want to use our framework for. We attach our debugger to the

Name	virtualized iDeFEND	userspace iDeFEND
Themida	✓	✗
VMProtect	✓	✗

Table 9.7: Detection of iDeFEND with Protected Applications

protected target application. Using all the presented tools with debugging protection, we are still able to install hardware breakpoints for the target location and extract data from the target application, as it halts at our breakpoints. Userspace debuggers, such as OllyDbg, caused the protected applications to terminate silently, as the protected binary detected the presence of the debugger. Table 9.7 summarizes the results of the detection. A checkmark means that our solution is not detected.

Briefly summarized, we evaluated the protection of our approach using state-of-the-art software protection mechanisms, which include debugging detection and modification protection. We made use of well-known anti-debugging tools to try and detect the framework presented in this section, with the result that the presented framework with virtualization extension is undetected.

Any form of introspection or debugging of an observation target introduces a certain amount of processing overhead. While this overhead is rather small in the userspace implementation, monitoring a virtualized target via Virtual Machine Introspection (VMI) introduces a considerable amount of overhead. A main cause for this virtualization overhead are VMExits, traps from the unprivileged machine into the hypervisor in order to execute privileged actions or due to debug events. In order to analyse the performance impact of monitoring through our iDeFEND extension, we setup two testing environments. We measured the execution time of CaS with and without monitoring by our extension.

In the first test we measured CaS execution time in PLINK, while in the second test the CaS execution time of PSFTP was measured. Both applications cause a considerable amount of VMExits through e.g. memory allocation and deallocation. We repeated the measurements 1000 times for with and without monitoring respectively.

In Figure 9.9 the results of our timing measurements are displayed. The Y-Axis represents the execution time of CaS in milliseconds [ms], the X-Axis represents an enumeration of the timing samples. Every measurement is displayed as a small cross. Blue crosses mark the execution time of an *unmonitored* CaS function, while green crosses mark the execution time of a *monitored* CaS function.

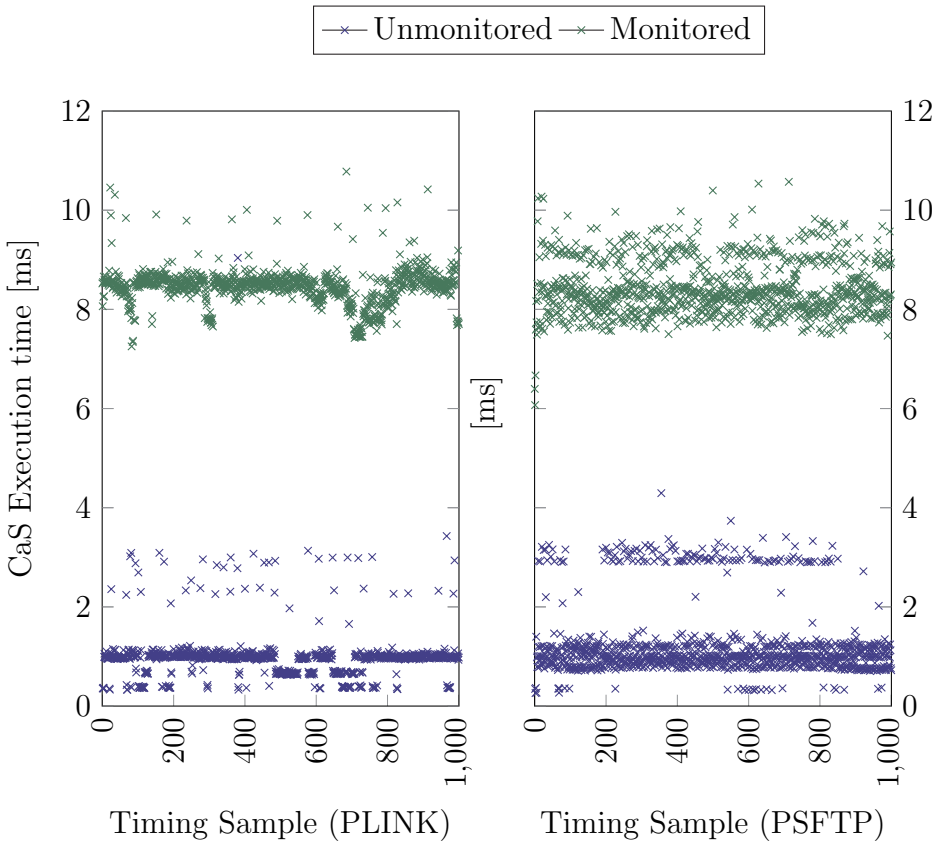


Figure 9.9: Performance Deviation through iDeFEND Monitoring

For PLINK, unmonitored execution of CaS took an average of 0.980827 *ms*. Due to the complexity of the PLINK implementation, incorporating memory allocation and deallocation in CaS, more traps into the hypervisor are performed. This results in a considerably higher average execution time of 9.118875 *ms* with monitoring.

Unmonitored execution of CaS in PSFTP took an average of 1.249789 *ms*. Likewise to PLINK, a high implementation complexity subsequently causes more traps into the hypervisor. Thus, the average CaS execution time with monitoring took 8.497180 *ms*.

The calculated average execution times are summarized in Table 9.8.

Name	Unmonitored execution in *ms*	Monitored execution in *ms*	Overhead in *ms*
PLINK	0.980827	9.118875	8.138048
PSFTP	1.249789	8.497180	7.247391

Table 9.8: Average CaS Execution with and without iDeFEND Monitoring

As shown in Table 9.8, monitoring through our iDeFEND extension introduces a considerable amount of processing overhead, related to the amount of guest VMExits during execution. However, the introduced overhead is not a constraint for the utilization of our iDeFEND extension. In the operational environments of our target applications, our framework monitors networking applications. Network communication itself often introduces significant amounts of delay, exceeding the presented overhead by far. Thus, despite the overhead, our extension is still entirely applicable for its intended use.

Chapter 10

Conclusion

Security investigation in encrypted environment has many challenges to be solved before we can talk about a practical solution. One major challenge is resulting from intrusion detection systems in encrypted traffic [72]. While different approaches try to do traffic analysis on the encrypted network data, others decrypt the traffic for further analysis. Analysis on encrypted traffic is not as efficient as traffic analysis on plaintext data. But decrypting the traffic can break the security of end-to-end encryption or it is a dedicated solution, which covers not all use cases.

For the case, no prevention is possible based on the network data, system-based protections can stop the attacker. Therefore, we examined the effectiveness of common protection mechanism to mitigate successful exploitation of vulnerabilities in applications. Taking the example of Format String Vulnerabilities, we considered beside system-based protections, e.g. Address Space Layout Randomization (ASLR) and Non Executable Bit (NX), also compiler-based protections, e.g. RELocation Read-Only (RELRO). Since remote Format String Attacks are mostly exploited using a memory leak, we elaborated an approach to take over the control of the remote application without any leakage, called Blind Attack. Common exploits rely on the presence of user controlled data on the stack and therefore we researched a method for writing the payload on the stack by having the user data only on the heap. Finally, we implemented a proof of concept to show the feasibility of our approach and proposed ideas how to protect against it.

Knowing the fact, that Format String Attacks can be detected easily based on the network data, the need for solutions to inspect the payload is given. In our thesis, we proposed a framework, which is called iDeFEND, to establish a data bridge between tools operating on plaintext traffic and the target application communication encrypted over the network. Our approach keeps up the end-to-end encryption and allows to extract the payload out of the encrypted network traffic. This informations allows an IDS to look for malicious content inside the network communication. Furthermore, our

framework is also able to intercept and modify the payload or to inject content created by the analyst into the data stream. Using these features, we provide a data bridge for security testing tools.

iDeFEND relies on designated functions inside binary applications. In common cases, we can detect them automatically, but in some specific situations manual reverse engineering would be required. Since this is a time-consuming process, we provide a framework for efficient function identification inside binary applications. In our dynamic approach, we enrich the application with meta data to have time-based state information. After processing the data we limit the number of executed functions and display them to the analyst. Using visual components, we help the analyst to identify the wanted function in a short time.

As iDeFEND can be used in a live environment to protect the system, we classified attacks against our framework and evaluated them. As a consequence, we used virtualization extensions to isolate iDeFEND from the operation system to protect it from dedicated attacks. To show the effectiveness we have implemented the framework and validated it using experiments on well-known applications.

Future Work

While the Attack Model introduced in Section 9.2 covers an extensive amount of iDeFENDs attack surface, some corner cases are explicitly not handled. iDeFEND, for example, is conceptually not designed to monitor designated malware, but rather operate as an intermediate interface between an observation target and an external entity. This external entity can be an IDS or an actor in a security testing environment. Defending against attacks from a dedicated malware observation target are therefore not part of the conceptual iDeFEND requirements. To investigate the encrypted network traffic of malware, the framework have to be hardened against these attacks.

Additionally, the iDeFEND extension is unable to detect attacks against cryptographic algorithms of the observation target. This is due to the fact, that the extension only inspects the payload of encrypted communication but not the communication itself. Thus, if the monitored application is overtaken through a malicious payload, sent in a cryptographic handshake, the extension does not notice. This can be solved by observing the encrypted data before it is getting decrypted. But this would require additional Breakpoints(BPs).

Finally, the implemented iDeFEND extension uses hardware BPs in order to coordinate VMI on the observation targets host VM. The current version of the XEN hypervisor does not support virtualization of ARM debug registers. However, due to the ARMv7.1 Debug Architecture Design, the overall number of breakpoints is tied to a range between

two and sixteen hardware breakpoint slots. In our testing environment, the available hardware provides a total of 6 HWBPs. Therefore the process matching is handled in the software logic of iDeFEND. Having multiple applications executing the same virtual address as the target process, leads to unnecessary VMexits and performance overhead. Future improvements of the XEN hypervisor, such as virtualization of ARM debug registers, would avoid VMexits from unmonitored applications.

List of Figures

2 Background

4 Blind Format String Attacks

5 Framework for Analysing Binary Applications

6 Security Testing of Mobile Applications

7 Protection using Virtual Machine Introspection

8 Function Identification

9 Evaluation

List of Tables

4 Blind Format String Attacks

6 Security Testing of Mobile Applications

8 Function Identification

9 Evaluation

Acronyms

A

ASLR Address Space Layout Randomization 27, 33, 36, 37, 39, 40, 43, 123

B

BP Breakpoint 48–51, 124

C

CA Certificate Authority 107
CaQ EnCrypt & EnQueue ..... I, 56–58, 66, 120, 121
CaS Crypt and Send 46, 48–52, 55, 60, 64, 66, 71, 72, 106, 120, 121, 125
CFG Control Flow Graph ..... I, 48, 49, 57, 58

D

DBI Dynamic Binary Instrumentation 82, 87, 88
DLL Dynamic Link Library 22, 51

E

EBP base pointer 37, 40, 41

F

FSA Format String Attack 25, 26, 33–37, 39, 40, 43, 109
FSV Format String Vulnerabilitie 33

G

GDB GNU-Debugger 32
GOT Global Offset Table 26, 37, 41
GUI Graphical User Interface 90, 119, 120

H

HDC Hypervisor Debug Component 78, 79
HDCR Hyp Debug Configuration Register 79
HIDS Host based Intrusion Detection System 31, 45, 46, 70, 79
HVM hardware virtual machine 71

I

IDS Intrusion Detection System . 30–33, 45, 46, 48, 52, 69, 70, 73, 76, 79, 106, 112, 115, 121, 123, 124
IP instruction pointer 52
IPC Inter-Process Communication 51, 52, 73

J

JOP Jump Oriented Programming 34

L

LSByte least significant byte 40, 41

M

MITM Man In The Middle 106, 110

N

NIDS Network based Intrusion Detection System 45, 106
NX Non Executable Bit 27, 33, 34, 36, 40, 43, 123

O

OS Operating System 48, 51, 52, 71–76, 78, 95

P

PLT Procedure Linkage Table 41
POC Proof Of Concept 39–41

R

RaD Receive and Decrypt .. I, 46, 48–52, 56–58, 60, 64, 66, 68, 71–73, 106, 125
RELRO RELocation Read-Only 26, 37, 40, 43, 123
ROP Return Oriented Programming 34, 38, 41

T

TEE Trusted Execution Environment 79
TOE Target Of Evaluation 84

V

VFA Virtual Function Address 72, 82, 84, 85, 87, 90–95, 98, 100, 101

VM Virtual Machine 71, 73–78, 121
VMI Virtual Machine Introspection 31, 32, 69, 71–75
VMM Virtual Machine Monitor 32, 75

Bibliography

[1] A10 and Ponemon institute. *Study: Uncovering hidden threats within encrypted traffic*. URL: `https://www.a10networks.com/sites/default/files/A10-EB-14106-EN.pdf` (visited on 06/21/2017).

[2] *A20 Datasheet, Revision 1.1*. Allwinner Technology. 4th Floor, B6 Building, NO.1, Software Road, Zhuhai, Guangdong Province, China, 2013.

[3] AA Abimbola, JM Munoz, and William J Buchanan. "NetHost-Sensor: Investigating the capture of end-to-end encrypted intrusive data". In: *Computers & security* 25.6 (2006), pp. 445–451.

[4] Payam Vahdani Amoli and Timo Hamalainen. "A real time unsupervised NIDS for detecting unknown and encrypted network attacks in high speed network". In: *Measurements and Networking Proceedings (M&N), 2013 IEEE International Workshop on*. IEEE. 2013, pp. 149–154.

[5] Anonymous. *Runtime Process Infection*. URL: `http://phrack.org/issues/59/8.html` (visited on 06/21/2017).

[6] *ARM Architecture Reference Manual, ARMv7-A and ARMV7-R edition*. ARM. 110 Fulbourn Road, Cambridge, England CB1 9NJ, 2012.

[7] M Augustin and A Baláž. "Intrusion detection with early recognition of encrypted application". In: *Intelligent Engineering Systems (INES), 2011 15th IEEE International Conference on*. IEEE. 2011, pp. 245–247.

[8] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. "SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr". In: *Information Security: 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006. Proceedings*. Ed. by Sokratis K Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 343–358.

[9] Arash Baratloo, Navjot Singh, and Timothy K Tsai. "Transparent Run-Time Defense Against Stack-Smashing Attacks." In: *USENIX Annual Technical Conference, General Track*. 2000, pp. 251–262.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the art of virtualization". In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 164–177.

[11] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. "Address obfuscation: An efficient approach to combat a broad range of memory error exploits". In: *Proceedings of the 12th USENIX security symposium*. Vol. 120. Washington, DC. 2003.

[12] Sandeep Bhatkar, Ron Sekar, and Daniel C DuVarney. "Efficient techniques for comprehensive protection from memory error exploits". In: *Proceedings of the 14th USENIX Security Symposium*. 2005, pp. 271–286.

[13] Aabha Biyani, Gantavya Sharma, Jagannath Aghav, Piyush Waradpande, Purva Savaji, and Mrityunjay Gautam. "Extension of SPIKE for encrypted protocol fuzzing". In: *Multimedia Information Networking and Security (MINES), 2011 Third International Conference on*. IEEE. 2011, pp. 343–347.

[14] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 30–40.

[15] Boost. *spsc_queue*. URL: `http://www.boost.org/doc/libs/1_64_0/doc/html/boost/lockfree/spsc_queue.html` (visited on 06/21/2017).

[16] Konstantin Böttinger, Dieter Schuster, and Claudia Eckert. "Detecting Fingerprinted Data in TLS Traffic". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '15. Singapore, Republic of Singapore: ACM, 2015, pp. 633–638.

[17] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies". In: *Black Hat* (2012).

[18] Derek Lane Bruening and Saman Amarasinghe. "Efficient, transparent, and comprehensive runtime code manipulation". PhD thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[19] Derek Bruening and Qin Zhao. "Practical Memory Checking with Dr. Memory". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 213–223.

[20] Derek Bruening, Qin Zhao, and Saman Amarasinghe. "Transparent Dynamic Instrumentation". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. New York, NY, USA: ACM, 2012, pp. 133–144.

[21] Bryan Buck and Jeffrey K Hollingsworth. "An API for Runtime Code Patching". In: *International Journal of High Performance Computing Applications* 14.4 (Nov. 2000), pp. 317–329.

[22] J Caballero, P Poosankam, C Kreibich, and Song D. "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering". In: *Proceedings of the 16th ACM conference on Computer and Communications Security*. 2009, pp. 621–634.

[23] Joan Calvet, José M Fernandez, and Jean-yves Marion. "Aligot : Cryptographic Function Identification in Obfuscated Binary Programs". In: *ACM Conference on Computer and Communications Security* (2012), pp. 2–4.

[24] Codenomicon. *Defensics*. URL: `https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html` (visited on 06/21/2017).

[25] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster. "Visual reverse engineering of binary and data files". In: *Visualization for Computer Security*. Springer, 2008, pp. 1–17.

[26] Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool*. URL: `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool` (visited on 06/21/2017).

[27] Intel Corporation. *Pin User Guide*. URL: `https://software.intel.com/sites/landingpage/pintool/docs/62732/Pin/html/` (visited on 06/21/2017).

[28] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. "FormatGuard: Automatic Protection From printf Format String Vulnerabilities." In: *USENIX Security Symposium*. Vol. 91. Washington, DC. 2001.

[29] Crispin Cowan, Steve Beattie, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. "PointguardTM: protecting pointers from buffer overflow vulnerabilities". In: *USENIX Security Symposium*. Vol. 91. Washington, DC. 2001.

[30] Cubietech Limited. *A20-Cubietruck-V1.0-130606, Technical Reference*. URL: `http://dl.cubieboard.org/model/CubieBoard3/Hardware/Board/A20_Cubietruck_HW_V10_130606.pdf` (visited on 06/21/2017).

[31] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization". In: *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM. 2013, pp. 289–298.

[32] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[33] DynamoRIO. *Dynamorio API*. URL: `http://dynamorio.org/docs/` (visited on 06/21/2017).

[34] DynamoRIO. *DynamoRIO History*. URL: `http://dynamorio.org/history.html` (visited on 06/21/2017).

[35] Claudia Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. De Gruyter Studium. De Gruyter, 2014. URL: `https://books.google.de/books?id=NUzoBQAAQBAJ`.

[36] M Eddington. *Peach Fuzzer*. URL: `http://www.peachfuzzer.com/` (visited on 06/21/2017).

[37] Stephen G Eick, Joseph L Steffen, and Eric E Sumner Jr. "Seesoft-a tool for visualizing line oriented software statistics". In: *Software Engineering, IEEE Transactions on* 18.11 (1992), pp. 957–968.

[38] Thales e-Security. *Encryption Application: Trends Study*. URL: `http://go.thales-esecurity.com/encryption-application-trends-study_16` (visited on 06/21/2017).

[39] J E Flood. *Telecommunication Networks*. IEE telecommunications series. Institution of Electrical Engineers, 1997.

[40] Vahid Aghaei Foroushani, Fazlollah Adibnia, and Elham Hojati. "Intrusion detection in encrypted accesses with SSH protocol to network public servers". In: *Computer and Communication Engineering, 2008. ICCCE 2008. International Conference on*. IEEE. 2008, pp. 314–318.

[41] Francesco Gadaleta, Yves Younan, Bart Jacobs, Wouter Joosen, Erik De Neve, and Nils Beosier. "Instruction-level countermeasures against stack-based buffer overflow attacks". In: *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*. ACM. 2009, pp. 7–12.

[42] Xiaofeng Gao, M Laurenzano, B Simon, and A Snavely. "Reducing overheads for acquiring dynamic memory traces". In: *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* 2005, pp. 46–55.

[43] Tal Garfinkel and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection". In: *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. 2003.

[44] Tal Garfinkel, Mendel Rosenblum, et al. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *NDSS*. Vol. 3. 2003, pp. 191–206.

[45] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols". In: *Security and Privacy in Communication Networks: 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*. Ed. by Bhavani Thuraisingham, XiaoFeng Wang, and Vinod Yegneswaran. Cham: Springer International Publishing, 2015, pp. 330–347.

[46] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. "Grammar-based Whitebox Fuzzing". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. New York, NY, USA: ACM, 2008, pp. 206–215.

[47] Vik Tor Goh, Jacob Zimmermann, and Mark Looi. "Experimenting with an intrusion detection system for encrypted networks". In: *International Journal of Business Intelligence and Data Mining* 5.2 (2010), pp. 172–191.

[48] Vik Tor Goh, Jacob Zimmermann, and Mark Looi. "Towards intrusion detection for encrypted networks". In: *Availability, Reliability and Security, 2009. ARES'09. International Conference on*. IEEE. 2009, pp. 540–545.

[49] Serge Gorbunov and Arnold Rosenbloom. "Autofuzz: Automated network protocol fuzzing framework". In: *IJCSNS* 10.8 (2010), p. 239.

[50] *GPG*. URL: `https://www.gnupg.org` (visited on 06/21/2017).

[51] Felix Gröbert, Carsten Willems, and Thorsten Holz. "Automated identification of cryptographic primitives in binary programs". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 6961 LNCS. 2011, pp. 41–60.

[52] Paul Haas. "Advanced format string attacks". In: *DEFCON 18* DEFCON 18 (2010).

[53] Hackmageddon. *Information Security Timelines and Statistics*. URL: `http://www.hackmageddon.com/2017/01/19/2016-cyber-attacks-statistics/` (visited on 06/21/2017).

[54] Dimitri van Heesch. *Doxygen*. URL: `http://www.doxygen.org/` (visited on 06/21/2017).

[55] Laurens Hellemons, Luuk Hendriks, Rick Hofstede, Anna Sperotto, Ramin Sadre, and Aiko Pras. "SSHCure: a flow-based SSH intrusion detection system". In: *IFIP International Conference on Autonomous Infrastructure, Management and Security*. Springer. 2012, pp. 86–97.

[56] SA Hex Rays. *IDA Pro Disassembler and Debugger*. URL: `https://hex-rays.com/products/ida/index.shtml` (visited on 06/21/2017).

[57] Alex Ho, Steven Hand, and Tim Harris. "PDB: Pervasive debugging with Xen". In: *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE. 2004, pp. 260–265.

[58] Intel. *Pin 3.0 User Guide*. URL: `https://software.intel.com/sites/landingpage/pintool/docs/76991/Pin/html/` (visited on 06/21/2017).

[59] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 325462-059US. 2016.

[60] Mathieu Jacomy, Sebastien Heymann, Tommaso Venturini, and Mathieu Bastian. "Forceatlas2, a continuous graph layout algorithm for handy network visualization". In: *Medialab center of research* 560 (2011).

[61] A Jaleel, M Mattina, and B Jacob. "Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads". In: *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* Feb. 2006, pp. 88–98.

[62] Sachin P Joglekar and Stephen R Tate. "ProtoMon: Embedded monitors for cryptographic protocol intrusion detection and prevention". In: *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*. Vol. 1. IEEE. 2004, pp. 81–88.

[63] Holger M Kienle and Hausi A Müller. "Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation". In: *Science of Computer Programming* 75.4 (2010), pp. 247–263.

[64] Fatih Kilic and Claudia Eckert. "iDeFEND: Intrusion Detection Framework for Encrypted Network Data". In: *Proceedings of the 14th International Conference on Cryptology and Network Security (CANS 2015)*. Vol. 9476. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 111–118.

[65] Fatih Kilic, Benedikt Geßele, and Hasan Ibne Akram. "Security Testing over Encrypted Channels on the ARM Platform". In: *Proceedings of the 12th International Conference on Internet Monitoring and Protection (ICIMP 2017)*. 2017.

[66] Fatih Kilic, Thomas Kittel, and Claudia Eckert. "Blind Format String Attacks". In: *Proceedings of the 10th International Conference on Security and Privacy in Communication Networks (SecureComm 2014)*. Vol. 153. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer International Publishing, 2015, pp. 301–314.

[67] Fatih Kilic, Hannes Laner, and Claudia Eckert. "Interactive Function Identification Decreasing the Effort of Reverse Engineering". In: *Proceedings of the 11th International Conference on Information Security and Cryptology (Inscrypt 2015)*. Springer International Publishing, 2016, pp. 468–487.

[68] Vladimir Kiriansky, Derek Bruening, and Saman P Amarasinghe. "Secure Execution via Program Shepherding". In: *Proceedings of the 11th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2002, pp. 191–206.

[69] T Kitagawa, M Hanaoka, and K Kono. "AspFuzz: A state-aware protocol fuzzer based on application-layer protocols". In: *Computers and Communications (ISCC), 2010 IEEE Symposium on*. 2010, pp. 202–208.

[70] Robert Koch and Gabi Dreo Rodosek. "Command evaluation in encrypted remote sessions". In: *Network and System Security (NSS), 2010 4th International Conference on*. IEEE. 2010, pp. 299–305.

[71] Robert Koch and Gabi Dreo Rodosek. "Security system for encrypted environments (S2E2)". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2010, pp. 505–507.

[72] Tiina Kovanen, Gil David, and Timo Hämäläinen. "Survey: Intrusion Detection Systems in Encrypted Traffic". In: *International Conference on Next Generation Wired/Wireless Networking*. Springer. 2016, pp. 281–293.

[73] Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. 2014.

[74] Thomas Leonhard. *Linux Kernel Fork*. URL: `https://github.com/talex5/linux` (visited on 06/21/2017).

[75] Zhuowei Li, Amitabha Das, and Jianying Zhou. "Theoretical basis for intrusion detection". In: *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*. IEEE. 2005, pp. 184–192.

[76] A R M Limited. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. ARM Limited, 2016, p. 5740.

[77] A R M Limited. *ARM big.LITTLE Technology*. URL: `http://www.arm.com/products/processors/technologies/biglittleprocessing.php` (visited on 06/21/2017).

[78] A R M Limited. *Procedure Call Standard for the ARM Architecture*. ARM Limited, Nov. 2015, p. 33.

[79] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vol. 40. PLDI '05 6. New York, NY, USA: ACM, 2005, pp. 190–200.

[80] John McCumber. "Information systems security: A comprehensive model". In: *Proceedings of the 14th National Computer Security Conference*. National Institute of Standards and Technology. 1991.

[81] Tilo Müller. "ASLR smack & laugh reference". In: *Seminar on Advanced Exploitation Techniques*. 2008.

[82] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[83] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[84] Nicholas Nethercote and Julian Seward. *Valgrind user manual*. URL: `http://valgrind.org/docs/manual/manual.html` (visited on 06/21/2017).

[85] T Newsham. "Format string attacks". In: *Guardent, Inc.* (2000).

[86] *OpenSSL*. URL: `https://www.openssl.org/` (visited on 06/21/2017).

[87] C Paar and J Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Berlin Heidelberg, 2009.

[88] Heidi Pan, Krste Asanović, Robert Cohn, and Chi-Keung Luk. "Controlling Program Execution Through Binary Instrumentation". In: *SIGARCH Comput. Archit. News* 33.5 (2005), pp. 45–50.

[89] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. "Pinpointing Representative Portions of Large Intel® Itanium® Programs with Dynamic Instrumentation". In: *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 37. Washington, DC, USA: IEEE Computer Society, 2004, pp. 81–92.

[90] Mathias Payer and Thomas Gross. "String Oriented Programming". In: *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. PPREW '13. ACM. 2013.

[91] Bryan D Payne. "Simplifying virtual machine introspection using libvmi". In: *Sandia report* (2012), pp. 43–44.

[92] Jonas Pfoh, Christian Schneider, and Claudia Eckert. "Nitro: Hardware-based system call tracing for virtual machines". In: *International Workshop on Security*. Springer. 2011, pp. 96–112.

[93] Planet. "A eulogy for format strings". In: *Phrack magazine* 14.67 (2010).

[94] A R M Holdings Plc. *Annual Report 2015: Strategic Report*. ARM Holdings plc, Aug. 2016, p. 64.

[95] Sergej Proskurin, Fatih Kilic, and Claudia Eckert. "Retrospective Protection utilizing Binary Rewriting". In: *14. Deutscher IT-Sicherheitskongress*. May 2015.

[96] Daniel Quist, Lorie M Liebrock, et al. "Visualizing compiled executables for malware analysis". In: *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*. IEEE. 2009, pp. 27–32.

[97] Dennie Reniers, Lucian Voinea, Ozan Ersoy, and Alexandru Telea. "The Solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product". In: *Science of Computer Programming* 79 (2014), pp. 224–240.

[98] Scut. *Exploiting Format String Vulnerability*. URL: `http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf` (visited on 06/21/2017).

[99] Beyond Security. *beSTORM Software Security Testing Tool*. URL: `http://www.beyondsecurity.com/bestorm.html` (visited on 06/21/2017).

[100] Tenable Network Security. *Nessus*. URL: `https://www.tenable.com/products/nessus-vulnerability-scanner/` (visited on 06/21/2017).

[101] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[102] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the effectiveness of address-space randomization". In: *Proceedings of the 11th ACM conference on Computer and communications security*. ACM. ACM Press, 2004, pp. 298–307.

[103] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert S Cohn, Kim M Hazelwood, Vladimir Vladimirov, and Moshe Bach. "Dynamic program analysis of Microsoft Windows applications." In: *ISPASS*. IEEE Computer Society, 2010, pp. 2–12.

[104] VMProtect Software. *VMProtect*. URL: `http://vmpsoft.com/products/vmprotect/` (visited on 06/21/2017).

[105] *SQLite*. URL: `http://sqlite.org` (visited on 06/21/2017).

[106] Robert Swiecki. *Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options*. URL: `https://github.com/google/honggfuzz` (visited on 06/21/2017).

[107] Simon Tatham, Owen Dunn, Ben Harris, and Jacob Nevins. *PuTTY: A free Telnet/SSH client*. URL: `https://www.chiark.greenend.org.uk/~sgtatham/putty/` (visited on 06/21/2017).

[108] PaX Team. *Homepage of PaX*. URL: `http://pax.grsecurity.net/` (visited on 06/21/2017).

[109] Oreans Technology. *Themida*. URL: `http://www.oreans.com/themida.php` (visited on 06/21/2017).

[110] The MITRE Corporation. *Common Vulnerabilities and Exposures*. URL: `https://cve.mitre.org/data/downloads/allitems.csv` (visited on 06/21/2017).

[111] Christopher Thompson, Maria Huntley, and Chad Link. *Virtualization detection: New strategies and their effectiveness*. URL: `https://people.eecs.berkeley.edu/~cthompson/papers/virt-detect.pdf` (visited on 06/21/2017).

[112] Philipp Trinius, Thorsten Holz, Jan Göbel, and Felix C Freiling. "Visual analysis of malware behavior using treemaps and thread graphs". In: *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*. IEEE. 2009, pp. 33–38.

[113] P Tsankov, M T Dashti, and D Basin. "SECFUZZ: Fuzz-testing security protocols". In: *Automation of Software Test (AST), 2012 7th International Workshop on*. 2012, pp. 1–7.

[114] Dan Upton, Kim Hazelwood, Robert Cohn, and Greg Lueck. "Improving Instrumentation Speed via Buffering". In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WBIA '09. New York, NY, USA: ACM, 2009, pp. 52–61.

[115] Amit Vasudevan and Ramesh Yerraballi. "Stealth Breakpoints". In: *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. 2005.

[116] Rui Wang, XiaoFeng Wang, Kehuan Zhang, and Zhuowei Li. "Towards Automatic Reverse Engineering of Software Security Configurations". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. New York, NY, USA: ACM, 2008, pp. 245–256.

[117] Haiquan Xiong, Zhiyong Liu, Weizhi Xu, and Shuai Jiao. "Libvmi: A library for bridging the semantic gap between guest os and vmm". In: *Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on*. IEEE. 2012, pp. 549–556.

[118] Akira Yamada, Yutaka Miyake, Keisuke Takemori, Ahren Studer, and Adrian Perrig. "Intrusion detection for encrypted web accesses". In: *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on*. Vol. 1. IEEE. 2007, pp. 569–576.

[119] D Yang, Y Zhang, and Q Liu. "BlendFuzz: A Model-Based Framework for Fuzz Testing Programs with Grammatical Inputs". In: *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. 2012, pp. 1070–1076.

[120] Michal Zalewski. *American Fuzzy Lop: a security-oriented fuzzer*. URL: `http://lcamtuf.coredump.cx/afl/` (visited on 06/21/2017).

[121] Mikhail Zolotukhin, Timo Hämäläinen, Tero Kokkonen, Antti Niemelä, and Jarmo Siltanen. "Data mining approach for detection of DDoS attacks utilizing SSL/TLS protocol". In: *Conference on Smart Spaces*. Springer. 2015, pp. 274–285.