GoldRusher: A Miner for Rapid Identification of Hidden Code

Aleieldin Salem
Technische Universität München
Garching bei München, Germany
salem@in.tum.de

***Abstract*—GoldRusher is a dynamic analysis tool primarily meant to aid reverse engineers with analyzing malware. Based on the fact that hidden code segments rarely execute, the tool is able to rapidly highlight functions and basic blocks that are potentially hidden, and identify the trigger conditions that control their executions.**

***Index Terms*—Reverse Engineering; Obfuscation; Hidden Code; Code Coverage; Malware**

I. Introduction

Code obfuscation is a common technique adopted by software vendors to protect their products from piracy, trampering, and reverse engineering [4]. Malware authors also utilize obfuscation to build malware that is (a) resilient against static analysis techniques, and (b) unintelligible for malware analysts attempting to manually study its behavior (e.g., to generate signatures for databases of antiviral software) [20]. To complement static analysis techniques, malware analysts and reverse engineers have been adopting dynamic methods that study the runtime behavior of malicious applications [11] [22]. In this context, malware authors started to adopt novel techniques that obfuscate–in addition to their appearance and structure–the runtime behavior of the malicious applications they implement [14]. We refer to those techniques as *behavioral obfuscation* techniques throughout this paper. Behavioral obfuscation can be categorized into two categories. On one hand, in what is usually refered to as *mimicry attack* [17], a malware instance can be implemented to seize the execution of its malicious payload, and mimic the behavior of benign applications. On the other hand, malicious applications can opt not to masquerade as benign ones, but rather surround their malicious behaviors with noise (e.g., via grafting their source code with irrelevant, bogus operations) [7] [13].

Techniques in either category can be further enhanced via *triggers*. In essence, triggers are conditional statements that control whether a malware is to execute its malicious code (i.e., payload). These conditional statements usually rely on a system property (e.g., current date and time), and vary according to the malware instance's purpose [12]. For example, in order to hinder dynamic analysis, some malware instances check for the nature of the underlying architecture (i.e., virtual or physical), prior to executing their behavior. Effectively, the malicious payload is *hidden*.

Finding hidden code segments and their corresponding triggers is, in theory, an undecidable problem [15] [18]. For instance, consider a trigger that fires whenever the program is about to halt. In fact, a recently discovered malware instance hijacks the shut down process of an Android device to trick the user into thinking the device is powered off, whilst spying on them using the camera [3]. Thus, using dynamic analysis tools to automatically find hidden code segments and their corresponding triggers can be reduced to the halting problem.

Given that manually attempting to find hidden code within malware instances cannot cope with the rates of malware release and discovery, there is a need for semi-automatic tools that can assist malware analysts rapidly find potentially hidden code and identify its triggers based on reliable empirical assumptions. The current tools that attempt to retrieve hidden code suffer from a number of limitations, though. Firstly, some tools require access to the source code of the application under test, which is difficult to acquire upon testing malicious applications. Secondly, to the best of our knowledge, the majority of hidden code extraction tools, such Renovo [16] and PolyUnpack [19], focus on one definition of hidden code (i.e., packed and encrypted code segments). Lastly, hidden extraction tools do not support the retrieval of triggers that control the execution of hidden code.

In this paper, we present *GoldRusher*[1], a semi-automatic reconnaissance tool agnostic to specific definitions of hiding code that quickly highlights code segments (i.e., functions and basic blocks), that scarcely execute and, thus, could be deliberately hidden. We argue that the outputs reported by GoldRusher can significantly shorten the analysis period by pointing analysts to suspicious code segments.

II. Hidden Code And Triggers

We define *hidden code* as code segments wrapped with boolean conditions that scarcely evaluate to the condition (i.e., true or false), that would execute the aforementioned code segments. The primary purpose of hiding code is to trick users into considering some code segments as non-existent. Code segments that infrequently execute (e.g., code to update a program), do not qualify as hidden code, because they are not deliberately hidden. We argue that deliberately hiding code segments rarely serves any normal, benign usecase and, thus, usually indicates malice. For example, it is difficult to picture

[1]https://github.com/aleisalem/GoldRusher

a usecase in which a legitimate program executes a segment of code only on April 1st at 12:00.

The functionality of the hidden code segments hinges on the malware author's intentions, which are dictated by the malware type (e.g., Keylogger, Ransomware, Adware, etc). Hence, we do not discuss the internal structure or possible functionalities of hidden code, and we only focus on their triggers. Triggers are conditional statements (e.g., if-statements), whose predicates usually evaluate to one boolean value in favor of the other. The predicates of those triggers compare values (e.g., acquired from user input or the system) with particular values specified by the author during implementation. Triggers can be categorized into three major categories according to the types and sources of the values checked by their predicates viz., temporal, secret-based, and environment-based triggers.

In temporal triggers, seen in figure 1a, the predicates check for temporal values (e.g., date/time). Depending on the implementation, the hidden code will start executing either on a particular time and date which is usually hard-coded into the program itself, or on a more dynamic, yet regular, condition such as "*every second Friday of the month*". This kind of triggers can be found in, both, malicious and benign applications. For example, commercial software that offers trial periods needs to check for the time and date before it executes code segments that disable some or all features of the software. Malicious applications utilize temporal triggers to execute their hidden, malicious code segments on a particular time/date; those instances are usually referred to as *Timebombs* [18].

As depicted in figure 1b, secret-based predicates compare values retrieved during run time with particular, secret values. Usually those secret values are stored in the form of a hash digest in order to prevent reverse engineers from retrieving them. In fact, Sharif et. al. make use of hash functions in hiding the value of their trigger conditions in their paper written to demonstrate the hazardous implications of behavioral code obfuscation [14].

Finally, environment-based trigger conditions focus on values describing the environment on which the program is running. Example values can be the current CPU architecture, the value of a specific Windows registry key, the type and version of the operating system, whether the program is running within a virtual environment, and so forth [12].

III. GoldRusher

A. Overview

The primary goal of our tool, GoldRusher, is to highlight to reverse engineers code segments that scarcely execute and, hence, are potentially hidden. Users of GoldRusher are expected to manually confirm the initial decisions made by the tool. In other words, we envision GoldRusher as a tool used within a semi-automatic setting. Moreover, the tool attempts to highlight the (type of) triggers–in accordance with the definitions in section II–that govern the execution of the potentially-hidden code segments.

As seen in figure 2, GoldRusher takes two main inputs viz., the path to the binary to analyze (i.e., target binary), and the types of the command-line arguments it expects. The tool currently supports only C/C++ binaries that have been compiled for *-nix based operating systems. The first step is to instrument the target binary with `printf` statements that imply whether a function or a basic block has executed. We use the tool *codeCoverage* [2] to instrument target binaries. The second phase of analysis includes executing the target binary multiple times. If the target binary expects command-line arguments, GoldRusher enables the user to either specify certain values that persist across all runs, or randomly generate values that belong to a certain type (e.g., `int`). For example, if a target binary sums two integers, the user can run GoldRusher with the values five and four, or with `int` and `int`. In the latter case, GoldRusher would generate random integers using the Python method `random.randint`. GoldRusher follows the same process to generate inputs expected by the binary during execution (e.g., via `scanf` or `stdin::cin`).

The third phase is responsible for parsing the print statements reported by the instrumented binary for every single execution. Using the information reported from the first three phases, we can figure out the number of times each function and basic block in the target binary has executed. The functions and basic blocks that have executed a number of times less than a *threshold* value are considered as hidden code candidates.

In the fourth phase, we use the tool *ltrace* [1] to retrieve the library calls that can be possibly made by the triggers. The same inputs used in the second phase are reused to execute the target binary using ltrace. The ouputs generated by ltrace are parsed and kept in memory in the fifth phase. Outputs from all executions are stored in a SQLite database for future reference. Lastly, outputs from the codeCoverage instrumented binary and ltrace are augmented to report (a) potentially hidden code segments, and (b) the triggers that control their execution. In the next sections we detail how the code segments are deemed as hidden, and how their triggers can be highlighted.

B. Hidden Code Extraction

We base our approach to identify and recover the hidden code segment on code coverage. Intuitively, hidden code segments will be executed (i.e., covered) the least number of times. Thus, if we execute a target binary with all possible input values it expects–whether the trigger's predicate is input-dependent or not–the hidden code segments will still be covered the least.

Based on this intuition, we can execute a target binary instrumented using codeCoverage with all possible input combinations, and record the code coverage results reported by the binary. The functions and basic blocks executed the least, particularly less than a threshold value specified by the user (e.g., 5%), are considered hidden. Nevertheless, it is infeasible to run a C/C++ program with all possible input combinations. For example, the total number of possible value combinations for a C/C++ program expecting two integers (4 bytes) is

```
time_t now = time(0);
tm *ltm = localtime(&now);
// some code
if((ltm->tm_mday == 29) ||
  ((ltm->tm_mon+1)% 2 == 0)){
    // execute hidden code
}
```

(a) Temporal trigger

```
std::string secret = "somesecret";
// some code
input = argv[3];
// more code
if (secret.compare(input) == 0){
    // execute hidden code
}
```

(b) Secret-based trigger

```
std::string result = exec("uname -a");
if(result.find("Fedora")
        == std::string::npos)
   // some code
else{
   // execute hidden code
}
```

(c) Environment-based trigger

Fig. 1: The three categories of triggers. In figure 1a, the hidden code is executed if the current system date is February 29^{th}. The hidden code, in figure 1b, executes if the fourth command-line argument used to run the program is *somesecret*. Lastly, in figure 1c, the hidden code will execute if the operating system on which the program is running is *not* the Linux distribution *Fedora*.

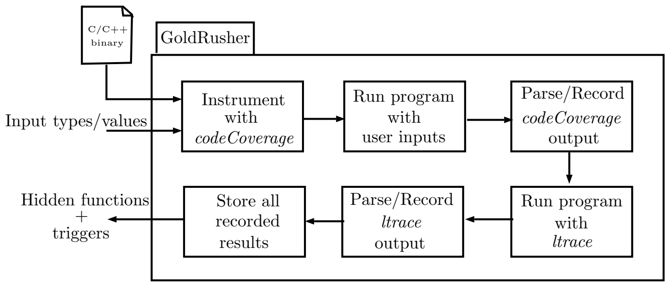
C/C++ binary
GoldRusher
Input types/values
Instrument with codeCoverage
Run program with user inputs
Parse/Record codeCoverage output
Hidden functions + triggers
Store all recorded results
Parse/Record ltrace output
Run program with ltrace

Fig. 2: An overview of GoldRusher's workflow and components.

$1.844674407 \times 10^{19}$ possibilities. In order to maintain the rapid aspect of revealing potentially hidden code segments, we need to sample the set of all possible inputs.

Consider all possible combinations of inputs (hereafter *test inputs*) as our population. If we consider the amount of code covered by each test case a continuous random variable (C), then C is expected to have a normal distribution. We argue that such normal distribution is a result of the fact that very few test inputs will cover small amounts of code, and very few test inputs will cover large amounts of code, perhaps including any hidden code segments. The majority of test inputs, however, are expected to cover medium amounts of code. Based on this assumption, we can be confident that a random sample of size ≥ 30 is representative of the entire population [6] [9].

In this context, we run the instrumented binary at least 30 times. However, users can opt to run the binary for an arbitrary number of times. After all runs, the functions and basic blocks that execute less than the threshold are labeled as potentially hidden.

C. Displaying Triggers

In section II, we discussed the different categories of triggers we focus on. Those triggers share something in common; upon realization of the condition that triggers the hidden code, they all use a variation of the `jmp` instruction to transfer control to the entry point of the hidden code. Recall that the previous phase highlights code segments that are potentially hidden. So, we can study the jump/branch instructions that transfer control to those segments. Disassembling the target binary is the straightforward method of retrieving `jmp`-like instructions. Nevertheless, anti-disassembly methods and code obfuscation can significantly hinder disassembling the code. Hence, we should rely on dynamically *tracing* the runtime behavior of the target binary.

In general, all types of traces are incapable of keeping track of branching statements, like `jmp`, because they are translated into instructions that run directly on the processor (i.e., they do not trigger any sort of system or library calls). With such limitation to the dynamic approach, one way to trace trigger conditions is to heuristically reason about the utilized library calls in the vicinity of the entry point of what is believed to be a hidden code segment based on the following facts and assumptions.

Firstly, triggers, which are essentially `jmp` statements, usually execute right before the execution of a hidden code segment. Secondly, as discussed in section II, triggers rely on values that need be retrieved, calculated, or compared using library calls (e.g., `strcmp`). Upon compilation, such library calls are expected to be executed prior to the jmp statement with the results of such calls being stored in some registers (e.g., `%eax`). Lastly, unlike the hidden code segments they call, triggers are expected to be executed on regular basis to check whether to execute their corresponding hidden segments.

The *-nix tool *ltrace* reports the library calls issued by a binary during runtime. It can keep track of the current value of the program counter during a call to a library function, as well. Using this information, we can display the names, arguments, and frequency of execution of the library functions next to their memory locations in the target binary. The user of GoldRusher can reason about the relation of such function calls to the code segments previously-highlighted as hidden based on, for instance, their proximity to such segments.

D. Display and Storage

The last phase of the analysis comprises displaying to the user the results gathered from codeCoverage, ltrace, and GoldRusher and storing them in a database for future study. The exact format of display and storage can be found in the tool's video demonstration. GoldRusher's output highlights the functions and basic blocks that execute the least, particularly a number of times less than or equal to a threshold specified by the user. The list of all functions and basic blocks retrieved by codeCoverage is displayed in green, and only the suspicious ones are displayed in red. Next to function names, we display the number of times they have been executed. The exact same is replicated for basic blocks and their instructions. In addition, we display the library calls made by such instructions during runtime, if applicable.

In the database, we store information about the target binary, such as its hash which can be matched against malware databases, its name, and the number of times it has been executed. We also store the functions retrieved from the target binary.For every single test input, we store the types and values of the arguments used to run the instrumented target binary along with the codeCoverage reports generated by the instrumented binary. Lastly, we store the path to the report displayed to the user at the end of analysis.

IV. Evaluation

A. Experiments

GoldRusher is built to enable reverse engineers to quickly reveal points of interest in binaries that might comprise deliberately hidden code. In this context, we evaluated the tool to assess (a) the reliability of its method and outputs, and (b) its efficiency. Unfortunately, it is difficult to acquire binaries (malicious or benign) that comprise hidden code segments. Thus, we used *Tigress*'s [10] `RandomFuns` option to generate 100 random programs that contain temporal and secret-based triggers.

Unlike temporal and secret-based triggers, environment-based triggers cannot be randomly generated, because they depend on values retrieved via specific commands/statements (e.g., `uname -a`). Furthermore, Tigress does not support this type of triggers. Consequently, we could not evaluate environment-based triggers in this set of experiments.

To test GoldRusher's performance against obfuscated code, we randomly obfuscated 50 of such programs using the Virtualization, Just-in-Time compilation, Control-Flow Flattening, and Function Splitting techniques.

The first experiment focuses on whether GoldRusher managed to highlight the hidden code segments embedded within a program. Within this context, a hidden code segment is a basic block that does not execute because its trigger never evaluates to true (e.g., because the provided password is incorrect). The second experiment solely focuses on the performance of GoldRusher, particularly the amount of time needed to complete testing an application.

B. Results

The results of the first experiment are depicted in table I. Using the Tigress-generated programs, we observed that GoldRusher could correctly highlight all the hidden code segments protected by temporal triggers along with their corresponding trigger conditions. Nevertheless, the tool's performance is lower against secret-based triggers. We argue that the tool is incapable of recognizing triggers that do not utilize library calls. For example, some programs may trigger hidden code segments if the sum of two integer inputs equals a specific number. Against such triggers, GoldRusher can only manage to highlight potentially hidden code segments.

TABLE I: GoldRusher's ability to reveal hidden code and triggers (30 test inputs per program, 5% threshold)

	Hidden Code Retrieval	Trigger Retrieval
Temporal Trigger	100%	100%
Secret-based Trigger	100%	$\approx$ 68%

As part of experiment 1, we generated another set of 100 programs using Tigress, 50 of which are obfuscated. However, this second set of programs did not include any checks or triggers, rather dummy loops that generate random integers. We ran GoldRusher against those programs to test whether it would mistakenly point out hidden code segments (i.e., false positives). We noticed that GoldRusher highlighted some basic blocks as potentially hidden. Upon manually inspecting them, we found out that such basic blocks checked whether the binary was executed with the correct amount of command-line arguments.

Mistakenly highlighting blocks as hidden is, unfortunately, inevitable. We argue that semit-automatically reverse engineering a binary is usually performed in stages, during which the reverse engineer is expected to tune the utilized tool (e.g., its inputs), to further discover segments of the binary. To conclude, the tool did not have any false positives on this sample dataset of programs. Needless to say, further experiments need to be conducted on larger datasets of real-world programs to further reinforce this conclusion.

Table II contains the time taken (in seconds) to run the Tigress-generated programs. The tool can complete the analysis including displaying and saving test inputs and coverage reports within 5 seconds. Needless to say, the generated programs are small in size with an average size of 68KB for clear, non-obfuscated programs and 71KB for their obfuscated counterparts. We reckon that malware instances are usually designed to be of small size as well to avoid raising suspicions. However, we plan on running GoldRusher on larger binaries in the future.

TABLE II: Time taken by GoldRusher to process Tigress-generated programs (30 test inputs per program, 5% threshold)

	Obfuscated Code	Clear Code
Temporal Trigger	5.264 (sec)	4.476 (sec)
Secret-based Trigger	5.361 (sec)	4.795 (sec)

V. Related Work

Previous efforts that attempt to retrieve hidden code focus on a particular definition of code hiding viz., packing. *Packing*, is a classic technique that transforms the code into a compressed, scrambled form. The transformation can range from a simple compression algorithm to using a public, or even proprietary, encryption algorithm [5].

Royal et. al. developed a tool, called *PolyUnpack* [19] that targets this particular breed of hidden code. The tool is based on a mixture of static and dynamic analyses of an application. Firstly, a reference control flow graph (CFG) of the application is statically generated. Such CFG is used by the dynamic analysis module of PolyUnpack to observe whether any of the code segments in the control flow graph are not executed.

The problem with the previous approach is that it heavily relies on the existence of a control flow graph of the analyzed code, which is generated based on a disassembled version of the program binaries. This leaves it vulnerable to obfuscation techniques that thwart static analysis. One of such techniques has been introduced by Linn et. al. [21].

To counter such problem, *Renovo* [16] drops the static aspect and adopts a fully-dynamic approach to the problem. It solely hinges on instrumenting memory writes during runtime assuming that instructions loaded into memory during runtime correspond to hidden segments of code. Renovo presumes that the packed code always dwells in the `.data` segment, and is unpacked, loaded, and executed during runtime. Nevertheless, as discussed earlier, hidden code can reside in within the `.code` segment, as well, as also demonstrated in [14].

Our tool, GoldRusher, shares some similarities with both tools. Similar to PolyUnpack, GoldRusher highlights functions and basic blocks that have not been executed during runtime, unlike Renovo which is concerned with highlighting hidden code segment upon being dynamically loaded. The tool, however, shares a similarity with Renovo in that it adopts a fully-dynamic approach to analyzing the program.

Unlike both tools, GoldRusher is agnostic to a specific code hiding technique, and attempts to highlight any potentially hidden code segments regardless of the technique used to hide them. Furthermore, the tool attempts to reveal any potential triggers that control the execution of such hidden code segments.

VI. Conclusion and Future Work

In this paper, we present GoldRusher, a tool meant to primarily aid reverse engineers semi-automatically analyze binaries to rapidly unveil deliberately hidden code segments. Given that GoldRusher is agnostic to specific definitions of code hiding and does not make presumptions about the intentions of the programs under test (i.e., malicious or benign), it can also be utilized by test engineers as a smarter tool for code coverage.

Our evaluation of the tool on a small dataset of randomly-generated, obfuscated programs shows potential of the tool being a helpful addition to a reverse engineer's toolset. However, we plan on evaluating the tool against larger datasets of (malicious) binaries. Evaluating the tool's performance on environment-based triggers, and thoroughly studying the false positives it reports is one of our priorities. Furthermore, we plan on enhancing the tool to be able to recognize triggers that do not rely on library calls. We also intend to link identified triggers with blocks they branch to. Lastly, we plan on releasing tutorials that demonstrate how GoldRusher can be used to reverse engineer real world (malicious) binaries.

References

[1] "ltrace(1) - Linux man page", 2017, https://linux.die.net/man/1/ltrace [Online]

[2] University of Wisconsin at Madison, "CodeCoverage", 2017, http://www.paradyn.org/html/tools/codecoverage.html [Online]

[3] AVG Now, "Malware is Still Spying On You Even When Your Mobile Is Off", 2015, https://now.avg.com/malware-is-still-spying-on-you-after-your-mobile-is-off/ [Online]

[4] Sebastian Banescu, Christian Collberg and Alexander Pretschner, "Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning", USENIX'17, 2017.

[5] Jason Raber, "Columbo: High Performance Unpacking", SANER'17, 2017, pp. 507-510.

[6] J. Toby Morkdoff, "The Assumptions of Normality", University of Iowa, 2016, http://www2.psychology.uiowa.edu/faculty/mordkoff/GradStats/part%201/I.07%20normal.pdf [Online].

[7] Sebastian Banescu, Tobias Wüchner, Aleieldin Salem, Marius Guggenmos, Martin Ochoa and Alexander Pretschner, "A framework for empirical evaluation of malware detection resilience against behavior obfuscation", MALWARE'15, 2015, pp. 40-47.

[8] Yueqian Zhang, Xiapu Luo and Haoyang Yin, "Dexhunter: toward extracting hidden code from packed android applications", ESORICS'15, 2015, pp.293-311.

[9] Stan Brown, "How Big a Sample Do I Need?", Brownmath, 2013, https://brownmath.com/stat/sampsiz.htm [Online].

[10] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra, "Distributed Application Tamper Detection via Continuous Software Updates", ACSAC'12, 2012, pp. 319-328.

[11] Manuel Egele, Theodoor Scholte, Engin Kirda and Christopher Kruegel, "A Survey on Automated Dynamic Malware-analysis Techniques and Tools", ACM Computer Survey'12, 2012, vol. 44, number 2, pp. 6:1-6:42.

[12] Michael Sikorski and Andrew Honig, "Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software", No Starch Press, 1 ed., 2012.

[13] Ilsun You and Kangbin Yim, "Malware Obfuscation Techniques: A Brief Survey", BWCCA'10, 2010, pp. 297-300.

[14] MI Sharif, A Lanzi, JT Giffi and W Lee, "Impeding Malware Analysis Using Conditional Code Obfuscation", NDSS'08, 2008.

[15] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song and Heng Yin, "Botnet Detection", Advances in Information Security'08, 2008, vol. 35, pp. 65-88.

[16] Min Gyung Kang, Pongsin Poosankam and Heng Yin, "Renovo: A Hidden Code Extractor for Packed Executables", WORM'07, 2007, pp. 46-53.

[17] Kevin Borders, Xin Zhao and Atul Prakash. "Siren: Catching evasive malware." Security and Privacy'06, 2006.

[18] Jedidiah R. Crandall and Gary Wassermann and Daniela A. S. Oliveira and Zhendong Su and S. Felix and Wu Frederic and T. Chong, "Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machine", Operating Systems Review'06, 2006, pp. 25-36.

[19] Paul Royal, Mitch Halpin, David Dagon, Robert Edmons and Wenke Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware", ACSAC'06, 2006, pp. 289-300.

[20] Arini Balakrishnan and Chloe Schulze, "Code Obfuscation Literature Survey", 2005 [Unpublished]

[21] Cullen Linn and Saumya Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly", CCS'03, 2003, pp. 290-299.

[22] Eleni Stroulia and Tarja Systä, "Dynamic Analysis for Reverse Engineering and Program Understanding", SIGAPP Application Comput. Rev.'02, 2002, vol. 10, no. 1, pp. 8-17.