Getting *RevEngE*: A System for Analyzing Reverse Engineering Behavior

Claire Taylor Christian Collberg
Department of Computer Science
University of Arizona
Tucson, AZ 85721
claire.g.taylor.1988@gmail.com, collberg@gmail.com

Abstract

Code obfuscation is a popular technique used by white- as well as black-hat developers to protect their software from Man-At-The-End attacks. A perennial problem has been how to evaluate the power of different obfuscation techniques. However, this evaluation is essential in order to be able to recommend which combination of techniques to employ for a particular application, and to estimate how long protected assets will survive in the field. We describe a system, RevEngE*, that generates random obfuscated challenges, asks reverse engineers to solve these challenges (with promises of monetary rewards if successful), monitors and collects data on how the challenges were solved, and verifies the correctness of submitted solutions. Finally, the system analyzes the actions of the engineers to determine the sequence and duration of techniques employed, with the ultimate goal of learning the relative strengths of different combinations of obfuscations.*

1 Introduction

Code Obfuscation seeks to protect valuable assets contained in software from those who have full access to it by making the software difficult to analyze. An obfuscator $\mathcal{O}$ applies a sequence of *code transformations* $\mathcal{T}_1 \circ \mathcal{T}_2 \circ \mathcal{T}_3 \circ \ldots$ to an input program P containing an asset a, transforming it into a program $P' = \mathcal{O}(P)$, such that P and P' are semantically identical, but extracting a from P' is more difficult than extracting a from P. Common assets include cryptographic keys, proprietary algorithms, security checks, etc. Obfuscating transformations are also used to induce *diversity*: given an original program P, a diversifying obfuscator generates a large number of semantically identical but syntactically different programs $\{P'_1, P'_2, \ldots\}$. Diversity is a defense against *class attacks*, where a single attack can successfully target all programs protected with a particular technique.

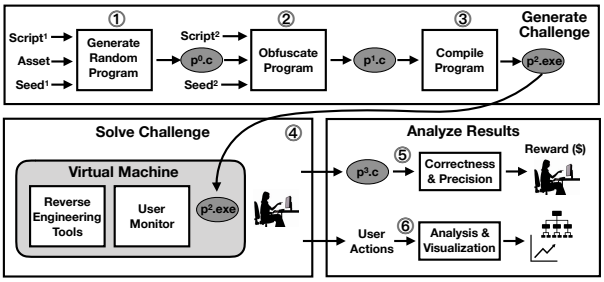

Figure 1: System Overview.

Attacks on obfuscation (known as *reverse engineering* or *deobfuscation*) aim to defeat the obfuscating transformations by extracting a close facsimile of the asset a from P'.

Much work has gone into developing methods to evaluate obfuscating transformations. Fundamental questions we may want to ask include (a) "how long will asset a survive in the field in a program P' that has been protected with obfuscating transform $\mathcal{T}$?"; (b) "how do transformations $\mathcal{T}_1$ and $\mathcal{T}_2$ compare with respect to their protective power and performance degradation?"; and (c) "what level of diversity can transformation $\mathcal{T}$ induce?"

Unfortunately, the validity of any theoretical models we develop to answer such questions will ultimately depend on how we define *the power of the adversary*.

In the project we report on here, we present a system, *RevEngE* (*Reverse Engineering Engine*), designed to build models from the behavior of reverse engineers. Figure 1 shows an overview of the system. Specifically, *RevEngE* generates a secret random program $\mathtt{p}^0.\mathtt{c}$ (point 1 in Figure 1) and transforms it with a variety of obfuscating transformations to a program $\mathtt{p}^1.\mathtt{c}$ (point 2) to create a collection of reverse engineering challenges. These challenge

programs are made available to reverse engineering experts anywhere to solve (point 3). Experts download the challenge p^2.exe into a virtual machine (point 4) equipped with a data collection subsystem that monitors and stores the engineers' behavior as they attempt the challenges. The reverse engineers submit a de-obfuscated solution p^3.c which is checked for correctness (point 5). To act as an incentive for accomplished reverse engineers to participate, substantial monetary awards are handed out to successful participants. The recorded user actions are analyzed (point 6) to create reverse engineering attack models, such as visualizations, trees, and Petri nets. The anonymized data set will be made available to the community for further analysis.

This paper is organized as follows. Section 2 discusses prior work on code obfuscation. Section 3 presents the design of *RevEngE*. Section 4 presents an evaluation of the system, and Section 5 concludes.

2 Related Work

We will next discuss prior work on obfuscation, obfuscation evaluation techniques, and the use of challenge problems to advance computer security.

2.1 Obfuscation

Obfuscating code transformations to protect assets in software date back at least to the early 1990s [17]. Progress was initially driven by the needs to protect cryptographic operations in Digital Rights Management systems [20, 12] and to hide algorithms in easily decompilable languages like Java [16]. Diversification also motivated engineers, both to protect operating system mono-cultures against attacks by malware [13], and, interestingly, to protect malware from detection [27].

The literature is rife with descriptions of obfuscating code transformations [15]. They typically fall in three categories: *control flow*, *data*, and *abstraction* transformations. Examples of control flow transformations include *control flow flattening* which removes existing control flow [33] and *opaque predicate insertion* which adds bogus control flow [16]. An extreme version of flattening is *virtualization* [36], in which the obfuscator generates a random virtual instruction set (V-ISA) for the target program P, translates P into this V-ISA, and generates a interpreter that executes programs in the V-ISA. Data obfuscations transform variables into a different representation, for example by encrypting them, xor:ing them with a random constant, or converting them into a representation based on Mixed Boolean Arithmetic [38]. Examples of abstraction transformations, finally, include class hierarchy flattening [18] and traditional optimizing transformations such as inlining, outlining, and loop unrolling.

2.2 Analyzing Obfuscation Resilience

There exist many obfuscating transforms and reverse engineering methods. Some are known through the academic literature, others are kept proprietary by purveyors of obfuscation tools, nation-states, and hackers. Furthermore, there are innumerable ways to combine such methods [19].

Evaluating the efficacy of strategies for obfuscation and reverse engineering has proven to be difficult: for a given obfuscation, many reverse engineering techniques might counter it, and for a given reverse engineering technique many obfuscations can render it ineffective.

2.2.1 Normative Evaluation

Some work attempts to reduce the complexity of an obfuscation technique to a known hard problem [16]. For example, it has been argued that opaque constants can be based on hard problems [31]. Unfortunately, this type of argument is fraught with issues. For example, just because a problem is hard in the worst case does not mean that a random instance will be, or that probabilistic algorithms cannot give an acceptable solution in reasonable amounts of time, or that the size of the problem will be sufficient. Additionally, assumptions made for normative evaluation—such as assuming static analysis only—often do not align with real world techniques. Finally, the human element and intuition is not captured by normative evaluation.

2.2.2 Software Metrics

It has been argued [16] that some software engineering metrics, such as control flow complexity, are related to the difficulty of reverse engineering an obfuscated program. However, this notion does not hold much promise, as empirical studies have shown no correlation between such metrics and human ability to reverse engineer [30].

2.2.3 Empirical Studies

Empirical analysis seeks to demonstrate the resilience of an obfuscation technique by testing it against reverse engineering attacks. Generally, empirical analysis focuses on two questions:

1. How well does an obfuscation technique resist an *automated attack*, using a particular tool or code analysis algorithm?
2. How well does an obfuscation technique resist a *human attacker* attempting to reverse engineer the protected code?

For the first question, researchers generate obfuscated code with an obfuscator and then run an automated tool on that code [2, 4, 3]. Success metrics include whether the automated tool extracted the protected asset or not, and the amount of resources (time and memory) consumed.

2.2.4 Human Studies

Ceccato et al. [8] argue that there is a "Need for More Human Studies to Assess Software Protection." Several such experiments have been carried out. In [10], Ceccato et al. conducted experiments to analyze the potency of an identifier renaming transform using Masters and PhD students as subjects. In [9] they expanded upon this earlier work with more students and more transforms. Viticchié et al. [32] conducted a similar experiment, wherein students were tasked with reverse engineering the *VarMerge* transform. In all of these experiments, the authors gave subjects particular reverse engineering tasks to accomplish on obfuscated pieces of software as well as a baseline, non-obfuscated piece of software for comparison. In each case, the authors found statistically significant differences in reverse engineering the code.

2.2.5 Need for Additional Studies

However, the number and type of subjects in the experiments limit their generalizability:

> students are probably not the best choice to model real subjects. Professional hackers could be better subjects to evaluate MATE[1] attacks exploitation, but it is considerably difficult to involve them [32].

Studies with only students or limited number of participants cannot be generalized to the field of professional/expert reverse engineers and hackers, yet in all of the previously cited experiments, the test subjects have been a limited group of students. As far as we are aware, Ceccato et al. [11] is the only published human subject reverse engineering study to employ professional experts, albeit with a small sample size.

Besides Ceccato's efforts, several challenges generated by the Tigress obfuscator [14] were hosted on the Tigress website[2]. Subjects (anyone on the Internet) were tasked with turning obfuscated code back into plaintext source and disclosing a description of their successful attacks. Several successful submissions by subject matter experts were submitted, which resulted in publications describing the reverse engineering methods used to defeat the obfuscations. Salwan et al. [26], for example, employed symbolic execution and taint analysis to defeat the obfuscation.

2.2.6 Current Methodology Limitations

The usefulness of human subject studies is also limited by the data collected. Previous efforts [11, 14] manually requested and analyzed subjects' reverse engineering strategies. However, conducting thorough analysis, particularly with high quality subjects, incurs large costs, drastically limiting scalability, both in terms of subject sample size and the diversity of the transforms tested.

3 Methodology

Our goal is to collect and analyze data about the strategies employed by reverse engineers. This data can be used to construct behavioral models which, in turn, can be used to evaluate the strength of particular obfuscating transformations against a variety of attacks. Ultimately, these insights can inform the design of new and better transformations and guide the deployment of current transformations.

To collect and analyze behavioral data, we must address 5 problems, detailed below.

3.1 Generate Obfuscation Challenges

As shown in Figure 1, the first step of *RevEngE* is to generate challenges by first generating random programs and then obfuscating these programs. The generation process can be described as a 5-tuple Gen:

$$Gen \quad = \quad (Script^1, Asset, Seed^1, Script^2, Seed^2) \quad (1)$$

Together, the elements of this tuple uniquely define the challenges we generate. Gen is kept confidential and it is the goal of the reverse engineer to recover the asset generated by Gen given only a compiled[3] obfuscated binary as input.

The generation of random programs takes 3 inputs: $Asset$ is the aspect of the program we want to protect (such as a security check, a license check, or the source code itself), $Script^1$ sets parameters on the language features to generate (number/size of functions, size of global and local state, kinds of control structures, etc), and $Seed^1$ can be varied to allow us to generate large numbers of unique programs with the same asset, and with the same parameters.

In the current version of *RevEngE*, we are asking reverse engineers to *de-obfuscate* a challenge, i.e. to transform an obfuscated binary program back into C code. In other words, the asset that they have to recover is the *source code* of the program itself. In a malware context, malware authors are generally not interested in protecting against complete code recovery. They may, however, be interested in protecting triggers or unpacking code from begin discovered or disabled using obfuscation. In future work we will therefore extend our challenges to include different types of assets, such as where the goal is to disable a security check or to recover an embedded cryptographic key.

Once programs have been generated, they are obfuscated (point 2 in Figure 1). This step takes 3 inputs: $\texttt{p}^0\texttt{.c}$ is the generated plaintext C program, $Script^2$ describes the se-

[1] MATE stands for *Man-At-The-End*.

[2] See `http://tigress.cs.arizona.edu/challenges.html`

[3] Source code challenges may be provided in the future as well, but compiled binaries tend to better match common real world use cases and thus are more appropriate.

quence of obfuscating transformations that should be applied to p^0.c, and $Seed^2$ can be varied to allow us to generate a diverse set of of uniquely obfuscated programs from the same obfuscation script.

3.1.1 Generating Random Programs

Generating random programs that are suitable targets for a reverse engineering challenge is a challenging research problem in itself. We extended the *Tigress* system's program generator to generate simple programs. Each generated function is, essentially, a hash function taking a list of numbers as inputs, and producing a list of numbers as output. A function conceptually consists of three phases: *expansion* which seeds a (large) state space from the (small) input, *mixing* which introduces control structures (if, while, switch) to update the state space, and *contraction* which reduces the state space to the output result.

To favor automated attacks, in *RevEngE* each challenge consists of multiple unique and obfuscated programs. To be considered successful, a reverse engineer has to solve all of them. In other words, in our current implementation a particular challenge $\mathcal{C}_i$ is generated by

$$\mathcal{C}_i = \overbrace{\{Gen_i, \ldots, Gen_i\}}^{n} \tag{2}$$

$$Gen_i = (script_i^{\text{gen}}, \_, rnd_1, script_i^{\text{obf}}, rnd_2) \tag{3}$$

where the rnd_j are pseudo-random numbers and where n, the number of obfuscated programs per challenge, is 10.

3.1.2 Obfuscating Programs

RevEngE uses an obfuscation tool (*Tigress*) that contains numerous types of obfuscating transformations: virtualization, self-modifying code, control flow flattening, function splitting and merging, adding dead code with opaque predicates, encoding expressions with mixed Boolean arithmetic, encoding data, encoding literal values, anti alias analysis, anti taint analysis, and hiding API calls. Each transformation can be customized with numerous variants and options.

As an example of the complexity introduced by obfuscation by virtualization, consider Figure 3 which shows the virtualized version of the program in Figure 2.

3.2 Recruiting and Rewarding Subjects

To participate, subjects download a virtual machine image,[4] install our monitoring system from `http://revenge.cs.arizona.edu`, and select and download one of our challenges. As part of the installation process users are also asked to give consent as required by the approval[5] granted by our institution's Institutional Review Board (IRB).

[4]Kali [23], Ubuntu, and Fedora are supported.

[5]University of Arizona IRB 1610963521.

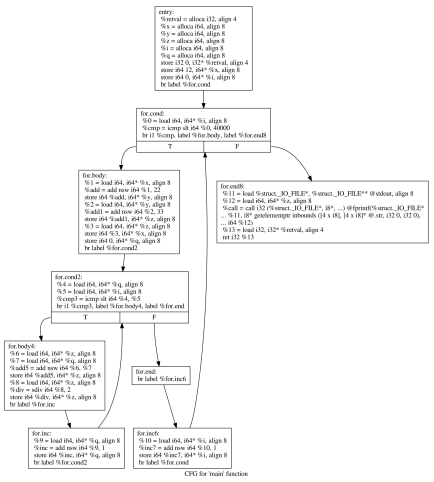

Figure 2: The control flow graph here derives from a simple, sample program with a single loop, a few arithmetic operations, and a print function at the end. LLVM [22] compiled this program and generated the control flow graph.

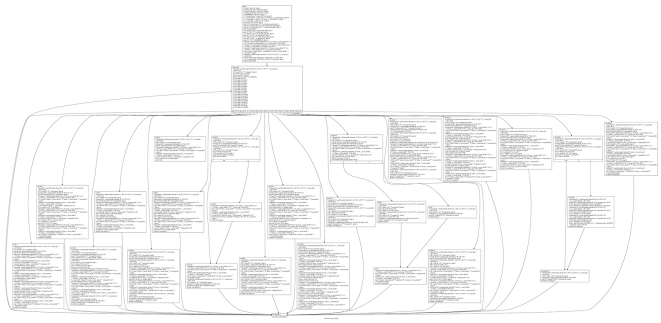

Figure 3: This control flow graph resulted from obfuscating the program in Figure 2 with a virtualization transformation.

At the present time, we have allocated USD 15,000 to be given out (in amounts of USD 100, 500, or 1,000) to users who submit successful solutions. The amounts are dependent on the perceived difficulty of each challenge.

Unlike previous studies which recruited subjects from student populations or professional red teams, we want to cast a wider net. There are several risks to the validity of the results of the study directly related to our ability to entice qualified reverse engineers to participate: it may be that our monetary rewards will be too small to attract interesting subjects, or that those with advanced skills will be unwilling to reveal them, or that clever adversaries will be able to identify ways to collect the reward without providing us with any useful attack data. It should be pointed out that our preliminary study (`tigress.cs.arizona.edu/challenges.html`) attracted multiple attackers both from industry and academia, even though the

rewards were minuscule. We therefore hope that the goals of *RevEngE* will provide some motivation to actors who have an interest in learning the results of the project.

3.3 Data Collection

Understanding reverse engineering behavior will require understanding of how a human interacts with a set of software tools. Such tools include debuggers, tracers, slicers, disassemblers, decompilers, symbolic analyzers, etc., both off-the-shelf tools and scripts developed by the human reverse engineer to aid in attacking a particular type of software protection. It is the goal of the *RevEngE*'s data collection framework to collect the interactions between reverse engineers, the tools that they use, and the challenge programs they are attacking. In order to comprehensively monitor the actions of a reverse engineer, *RevEngE* monitors

- processes and events such as file open/close and process creation (including arguments);
- window focus changes and data such as x/y-location, size, and title;
- mouse events and data such as x/y-location and event type (move, press, ...);
- any keyboard key presses.

Finally, a screen shot is taken every 15 seconds to collect data not otherwise captured. Every event is timestamped and relationships between events are recorded, such as keyboard and mouse input occurring within a particular in-focus window.

3.4 Correctness and Precision of Solutions

When users submit a proposed solution to a reverse engineering challenge ($\mathtt{p}^3.\mathtt{c}$ in Figure 1), we must determine whether it is, in fact, an acceptable solution. Two properties of $\mathtt{p}^3.\mathtt{c}$ must be tested:

- Is $\mathtt{p}^3.\mathtt{c}$ a *correct* solution, i.e. does it have the same behavior as the original program $\mathtt{p}^0.\mathtt{c}$?
- Is $\mathtt{p}^3.\mathtt{c}$ a *precise* solution, i.e. was the user able to remove the obfuscation from $\mathtt{p}^2.\mathtt{exe}$, such that $\mathtt{p}^3.\mathtt{c}$ is close to the original program $\mathtt{p}^0.\mathtt{c}$?

A malicious user could submit a program that is simply a decompiled (but not de-obfuscated) version of the challenge program, claiming that it solves the challenge. Such as "solution" may be correct, but it is not precise. Or, they may submit a de-obfuscated program that works for most inputs but fails for some corner cases. Such a solution may be precise, but it is not correct.

3.4.1 Correctness

RevEngE checks the solution correctness by *testing*: The submitted program is run inside a security sandbox (firejail [1], in our case), on a set of chosen inputs, and the output is compared to that of the original program. This is easy to do, since the challenge programs are simple hash functions that read input numbers from the command line, and print output numbers on standard output. The issue is how to select a set of comprehensive input test cases that will distinguish a correct from an incorrect proposed solution.

We use two methods for generating test cases. First, we generate a large number of random inputs $\mathcal{I}_{\mathrm{rnd}}$ taken from some distribution. Unfortunately, there is no guarantee that all paths of the program will be covered by these inputs.

The second technique uses symbolic execution. We run Klee [7] on the original program $\mathtt{p}^0.\mathtt{c}$ to determine a set of inputs $\mathcal{I}_{\mathrm{Klee}}$ which covers as many execution paths as possible. In order to run Klee, the program's argument variable is annotated as symbolic within the code, and Klee's default search is run for 7 hours on the resulting program. We found that few to no additional test cases were found after this period of time and have not had success finding additional test cases with other Klee search algorithms at this time.

We then merge the input sets $\mathcal{I}_{\mathrm{rnd}}$ and $\mathcal{I}_{\mathrm{Klee}}$, and run the program $\mathtt{p}^0.\mathtt{c}$ on these inputs through a *source code coverage tool* (`gcov` in our case [6]) to determine what fraction of the paths are covered by our test cases.

3.4.2 Precision

To evaluate the quality of a proposed solution we need to measure how similar $\mathtt{p}^3.\mathtt{c}$ is to the original program $\mathtt{p}^0.\mathtt{c}$. This could be done statically, for example by comparing the programs' control flow graphs, as was done by Krienke [21] by comparing program subgraphs for similarity. However, a perfectly acceptable solution might be structurally very different from the original: the reverse engineer might have unrolled loops, for example, or inlined functions. In such cases static code similarly measures would fail.

RevEngE instead compares the execution behavior of the original and de-obfuscated programs. Even then, we cannot expect a deobfuscated challenge to execute identically to the original program, therefore our precision metric has to be *fuzzy*. Thus, *RevEngE* uses a custom-built, LLVM based instruction counter to track the number of instructions executed, and we classify these into groups such as *arithmetic*, *load/store*, *branch*, etc. These sets of instructions are then compared to $\mathtt{p}^0.\mathtt{c}$'s performance for similarity. This provides a deterministic comparison of high-level LLVM instructions executed, allowing a reliable program profile to be generated, similar to work done by Neustifter [24].

3.5 Analyze and Visualize User Actions

The outcome of our reverse engineering challenges is data that describes how each user attacked each obfuscated program in each challenge, and how successful their attack was. Conceptually, our data consists of a list of tuples

$$(user, challenge, program, correctness, precision, actions).$$

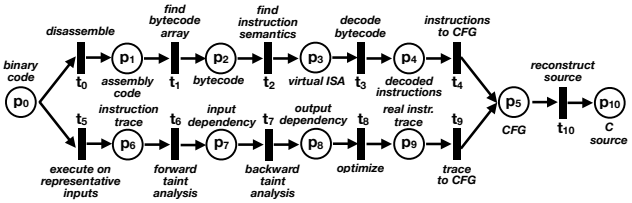

Figure 4: Petri net model of two de-virtualization de-obfuscation analyses.

For example, the following user action data

$$\langle(u_0, \mathcal{C}_1, 4, \mathit{correct}, 0.9, \langle a_0, a_1, \ldots\rangle), \quad (4)$$

$$(u_1, \mathcal{C}_3, 2, \mathit{correct}, 0.7, \langle b_0, b_1, \ldots\rangle), \ldots\rangle \quad (5)$$

shows that user u_0 attempted to reverse engineer the 4th program in challenge $\mathcal{C}_1$ (each challenge contains 10 individual programs to attack, see Section 3.1.1), our correctness checks indicate that their attack succeeded, the precision of the de-obfuscation was 0.9 (see Section 3.4.2), and the user went through the sequence of actions $\langle a_0, a_1, \ldots\rangle$. Each user action a_i is a tuple $(\mathit{time}, \mathit{kind}, \mathit{value})$ where time is the timestamp when the action was recorded, kind describes the type of data that was collected (such as screenshot, mouse movement, keyboard input, etc.; see Section 3.3), and value is the data collected. An example action list might look like this:

$$\langle(10{:}23.04, \mathit{screenshot}, \texttt{1.jpg}),$$
$$(10{:}23.05, \mathit{mouseMove}, (330, 450)),$$
$$(10{:}23.10, \mathit{fileOpen}, \texttt{angr}), \ldots\rangle$$

3.5.1 Building Reverse Engineering Models

Ultimately, our goal is to be able to extract reverse engineering models from the collected data. While such models have been constructed qualitatively by hand in the past, our data will allow them to be based on the actual actions of reverse engineers in the field.

Common graphical ways to model attacks are *attack trees* and *Petri nets* [29, 37, 5, 34]. As an example, consider Figure 4 which shows a Petri net modeling two types of attack on virtualized code. The top path $\langle p_0, t_0, p_1 \ldots, t_4, p_5, t_{10}, p_{10}\rangle$ represents the *static analysis* proposed by Rolles [25] and the lower path $\langle p_0, t_5, p_6 \ldots, t_9, p_5, t_{10}, p_{10}\rangle$ represents the *dynamic analysis* proposed by Yadegari et al. [35]. The t_i are *transitions*, actions performed by reverse engineers, and the p_i are *states*. A successful attack is a path from p_0 to the terminal state, in this case p_{10}, which represents the acquisition of de-obfuscated source code.

Thus, an attack Petri net is a 6-tuple [34]

$$(\mathit{states}, \mathit{transitions}, \mathit{arcs}, \mathit{paths}, \mathit{rate}, \mathit{cost})$$

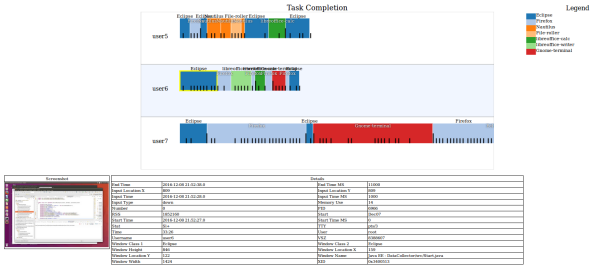

Figure 5: The visualization component for *RevEngE*'s data collection consists of a Gantt chart of in-focus windows and task events per user session; selecting a rectangle shows additional information such as screenshots and the process information.

where rate represents the rate of attack progress, and $\mathit{cost}(t_i)$ represents the cost (to the attacker) of taking a particular transition t_i. It is the goal of *RevEngE* to examine the collected data and recover states, $\mathit{transitions}$, and arcs from the sequence of actions that the reverse engineer has performed, and to recover rate and cost by examining the time and memory consumed performing these actions.

3.5.2 Visualization of User Actions

Figure 5 shows the visualization component of *RevEngE* which will allow us to interactively explore the action events and manually build the attack graph. The visualization presents events along a Gantt chart timeline with current task information presented below. Ultimately, we would like to automatically extract Petri net attack graphs from the collected user action data. We are currently investigating how to combine sequence and time series analyses, machine learning classification techniques, and additional visualization techniques to accomplish this.

4 Evaluation

To be successful, *RevEngE* needs to be able to properly evaluate submitted programs, and the overhead of data collection must not discourage reverse engineers from using the system. We evaluate these issues next.

4.1 Correctness analysis

To determine whether a submitted program in fact solves a challenge, we need to determine that it is equivalent to the original, generated, random program. We use two methods to generate test cases to compare the two programs for equivalence: symbolic analysis using Klee, and random generation from a distribution. Figure 6 shows the path coverage that these methods achieve.

On average, we achieve 85% coverage. In the future we will remedy this by a) using multiple symbolic analysis engines which use different search strategies, b) extend-

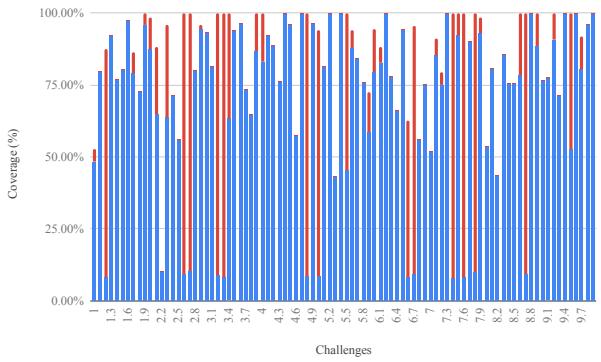

Figure 6: Path coverage (using `gcov`) for generated test cases. Blue bars represent the number of unique test cases contributed by Klee's symbolic analysis, red bars unique test cases contributed by random input generation.

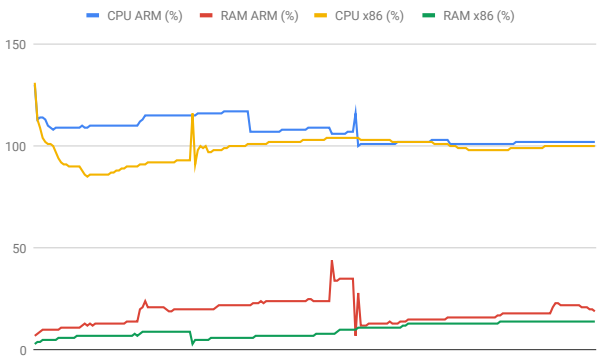

Figure 7: CPU and RAM usage for two data collection runs, one on an x86 device and one on an ARM device, both using an Ubuntu virtual machine.

ing symbolic analysis with *fuzzing* [28]. Currently, when we still fail to achieve 100% coverage, we re-generate the random challenge program.

4.2 Overhead of data collection

RevEngE's data collection software must be relatively transparent, such that it does not interfere with subjects' ability to reverse engineer. In our experiments, the data collection occupied about 120% of a single x86 core CPU and 40% of 4 GB of memory after executing for about 1.68 days. Figure 7 shows an ARM device that was allocated 2 GB of memory and left executing for 1.2 hours reaching 102% of a single core usage and 20% of memory usage. The device was significantly laggy—though usable—compared to the x86 virtual machine with additional resources.

After executing for 1.68 days on a high resolution (1920x1440) display, the initial x86 data collection trial run had accumulated about 1 GB of screenshot data, and another 400 MB of process data—these two polled resources dwarf all other data by at least an order of magnitude.

5 Conclusion

The field of code obfuscation and reverse engineering lacks confidence in methods. Where many fields have demonstrated methods' efficacy through challenges of various types—particularly cryptography with the RSA Challenge—no such effort has demonstrated the hardness of different obfuscation techniques and, inversely, the strength of reverse engineering methods to counter them.

In this paper we introduce the *RevEngE* competition which generates reverse engineering challenges. Using novel data collection and auto-grading capabilities, *RevEngE* promises to efficiently and objectively evaluate participants' methods to defeat code obfuscations.

Sharing Statement: The *RevEngE* system can be accessed at `http://revenge.cs.arizona.edu`. All the source code is freely available at `https://github.com/cgtboy1988/RevEngE`. The binary for *Tigress* used to generate challenges can be downloaded from `http://tigress.cs.arizona.edu` (source code is available to researchers on request). Properly anonymized data generated by the system will be made freely available to the research community.

This work was supported by NSF award 1525820.

References

[1] Firejail Security Sandbox.

[2] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200. ACM, 2016.

[3] S. Banescu, M. Ochoa, and A. Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 45–51. IEEE, 2015.

[4] S.-E. Banescu. *Characterizing the Strength of Software Obfuscation Against Automated Attacks*. PhD thesis, Technische Universität München, 2017.

[5] C. Basile, D. Cavanese, L. Regano, P. Falcarin, and B. De Sutter. A meta-model for software protections and reverse engineering attacks. *Journal of Systems and Software*, 2019.

[6] S. Best. Analyzing code coverage with gcov. *Linux Magazine*, pages 43–50, 2003.

[7] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[8] M. Ceccato. On the need for more human studies to assess software protection. In *Workshop on Continuously Upgradeable Software Security and Protection*, pages 55–56, 2014.

[9] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074, 2014.

[10] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 178–187. IEEE, 2009.

[11] M. Ceccato, P. Tonella, C. Basile, B. Coppens, B. De Sutter, P. Falcarin, and M. Torchiano. How professional hackers understand protected code while performing attack tasks. In *Proceedings of the 25th International Conference on Program Comprehension*, pages 154–164. IEEE Press, 2017.

[12] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot. White-box cryptography and an aes implementation. In *9th Annual Workshop on Selected Areas in Cryptography (sac 2002.*, 2002.

[13] F. B. Cohen. Operating system protection through program evolution. *Computer Security*, 12(6):565–584, 1993.

[14] C. Collberg. The Tigress C Diversifier/Obfuscator.

[15] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2010.

[16] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[17] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski. Guest editors' introduction: Software protection. *IEEE Software*, 28(2):24–27, 2011.

[18] C. Foket, B. De Sutter, B. Coppens, and K. De Bosschere. A novel obfuscation: Class hierarchy flattening. In *5th International Symposium on Foundations & Practice of Security (FPS 2012)*, number 7743 in LNCS, pages 194–210, 10 2012.

[19] K. Heffner and C. S. Collberg. The obfuscation executive. In *Information Security, 7th International Conference*, pages 428–440. Springer Verlag, 2004. Lecture Notes in Computer Science, #3225.

[20] J. J. Horning, W. O. Sibert, R. E. Tarjan, U. Maheshwari, W. G. Horne, A. K. Wright, L. R. Matheson, and S. Owicki. Software self-defense systems and methods, Sept. 2005. Assigned to InterTrust Technologies Corporation.

[21] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.

[22] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[23] J. Muniz. *Web Penetration Testing with Kali Linux*. Packt Publishing Ltd, 2013.

[24] A. Neustifter. *Efficient profiling in the LLVM compiler infrastructure*. na, 2010.

[25] R. Rolles. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 1–1. USENIX Association, 2009.

[26] J. Salwan, S. Bardin, and M.-L. Potet. Symbolic deobfuscation: From virtualized code back to the original. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 372–392. Springer, 2018.

[27] F. Skulason. The mutation engine–the final nail? *Virus Bulletin*, Apr. 1992.

[28] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*. The Internet Society, 2016.

[29] Q. Su, F. He, N. Wu, and Z. Lin. A method for construction of software protection technology application sequence based on petri net with inhibitor arcs. *IEEE Access*, 6:11988–12000, 2018.

[30] I. Sutherland, G. E. Kalb, A. Blyth, and G. Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 25(3):221–228, 2006.

[31] R. Tiella and M. Ceccato. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 182–192. IEEE, 2017.

[32] A. Viticchié, L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella, and R. Tiella. Assessment of source code obfuscation techniques. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, pages 11–20. IEEE, 2016.

[33] C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, University of Virginia, Oct. 2000. `http://citeseer.ist.psu.edu/wang00security.html`.

[34] H. Wang, D. Fang, N. Wang, Z. Tang, F. Chen, and Y. Gu. Method to evaluate software protection based on attack modeling. In *IEEE 10th Int'l Conf. on High Performance Computing and Communications*, pages 837–844, Nov. 2013.

[35] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 732–744. ACM, 2015.

[36] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 674–691, May 2015.

[37] G. Zhang, P. Falcarin, E. Gómez-Martínez, S. Islam, C. Tartary, B. De Sutter, and J. d'Annoville. Attack simulation based software protection assessment method. In *Cyber Security And Protection Of Digital Services (Cyber Security), 2016 International Conference On*, pages 1–8. Ieee, 2016.

[38] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *Proceedings of the 8th International Conference on Information Security Applications*, WISA'07, pages 61–75, Berlin, Heidelberg, 2007. Springer-Verlag.