



UNICA

UNIVERSITÀ
DEGLI STUDI
DI CAGLIARI



Università di Cagliari

UNICA IRIS Institutional Research Information System

This is the author's accepted version of a contribution presented at the First International Workshop, CyberSec4Europe 2022 Venice, Italy, April 17–21, 2022.

When citing this work, please cite the original published paper:

Daniele Canavese, Leonardo Regano and Antonio Lioy, "Computer-Aided Reverse Engineering of Protected Software", in **Digital Sovereignty in Cyber Security: New Challenges in Future Vision**, Springer, 2023, pp. 3-15.

The publisher's version is available at:

https://doi.org/10.1007/978-3-031-36096-1_1

Computer-Aided Reverse Engineering of Protected Software

Daniele Canavese^(✉), Leonardo Regano, and Antonio Lioy

Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy
{daniele.canavese,leonardo.regano,antonio.lioy}@polito.it

Abstract. Reverse engineering is undoing or circumventing the protections deployed on a code region. Software crackers perform this to remove license checks in commercial applications and video games, but it can also be done for legitimate purposes. Many software houses perform a security assessment phase by reverse engineering their protected software before releasing it to the market. Furthermore, anti-virus experts need to reverse engineering malware (e.g., viruses and ransomware) to understand how it works and spreads. Typically, reverse engineering is performed by hand with minimal computer support with debuggers, decompilers, and disassemblers. Nevertheless, in recent years, new research directions have proposed various promising automatic methods, primarily based on machine learning and symbolic execution techniques.

Keywords: reverse engineering · software protection · software obfuscation · machine learning · natural language processing · symbolic execution · concolic execution

1 Introduction

Software protection techniques are one of the cornerstones of modern (commercial) applications. Their flexibility and applicability are endless. Commercial software frequently uses them to fight piracy and to delay the release of cracks that can significantly reduce the monetary gain of software houses. Several video games are also protected to avoid cheaters. In addition, these techniques are also frequently used by various malware to fool anti-viruses.

Among the vast plethora of protection techniques, obfuscation techniques are the most widely used ones. The key idea of these algorithms is to boost the code complexity so that the effort necessary by an attacker to remove it is so high that it becomes impractical or economically unworthy. Many research papers showed that obfuscations, and software protection techniques, in general, increase the attack time by using empirical human studies [4, 7, 22, 23]. Increasing the code complexity can be performed in various ways, such as rewriting mathematical expressions with bigger ones, introducing fake branches and convoluted loops, or leveraging odd pointer arithmetics to perform computations.

With the term *reverse engineering*, we indicate removing or bypassing the protections applied to a piece of code. This is usually done at the binary level by inspecting the assembly instructions. Attackers reverse engineer software for piracy-related purposes, such as cracking a commercial application or a video game. On the other hand, these techniques can also be used for defensive purposes. For instance, several software houses, before releasing a new protected software on the market, frequently perform several reverse engineering tests to assess the strength of its protections. Furthermore, several malware uses various protection techniques to thwart anti-virus detection.

Currently, the job of the reverse engineer is performed mainly by hand, with minimal automated support. Binaries are usually dissected via disassemblers and debuggers, at most with some custom scripts, and this process requires a high level of expertise and a long time. In the last few years, a new trend has started to appear gradually: automating most, if not all, the operations performed by a reverse engineer. Even if this area is still in its infancy, recent progress in machine learning and symbolic execution has already started to provide promising results. In the following sections, a variety of recent papers, patents, technologies, and tools will be studied. The goal of this paper is twofold. On one side, it is an interesting survey on the most recent automatic reverse engineering trends. On the other, it may serve as food for thought for researchers and experts willing to enter this fascinating area of cybersecurity.

The works presented in the following paragraphs suggests that replacing (for the most part) the human being in performing this kind of analysis is very feasible. Shortly, it is very likely that we will see a large adoption of AI-based techniques to speed up the protected area identification of released applications and libraries. It is foreseeable that machine-learning approaches will be prime citizens in this field, especially based on natural language processing techniques.

2 Protected Region Identification

When reverse engineering a binary file (e.g., an application, a plug-in, or a library), the first step is usually to identify the protected areas, as this is a strong indication that they may be the resources the attacker is looking for (for example, a license check function that needs to be cracked). This is traditionally performed by hand with the aid of disassemblers and debuggers, making it a time-consuming and labor-intensive operation. However, recent developments have shown that AI-powered methods can be used successfully to parse a binary file and detect protected regions quickly.

The critical idea in being able to identify a protected area is that software protections tend to change the code by leaving a consistent and recognizable pattern known as a *fingerprint* [16]. For instance, Fig. 1 shows an example of a simple function written in the C programming language protected with the control flow graph flattening obfuscation technique [12]. This technique transforms the code into a giant while loop with several nested branches, which can be identified by inspecting the (source and assembly) code or other representations such as the control flow graph.

```

1 x = y + z;
2 if (x < 0 || x > 100)
3     w = c1;
4 else
5     w = c2;
6

```

(a) Pre-protection code.

```

1 i = 0;
2 while (1) {
3     if (i == 0) {
4         x = y + z;
5         i = 1;
6     } else if (i == 1) {
7         if (x < 0 || x > 100)
8             i = 2;
9         else
10            i = 3;
11    } else if (i == 2) {
12        w = c1;
13        i = 4;
14    } else if (i == 3) {
15        w = c2;
16        i = 4;
17    } else if (i == 4) {
18        break;
19    }
20 }
21

```

(b) Post-protection code.

Fig. 1. Control flow flattening obfuscation example.

2.1 Security Assessment

The software protection field research mainly focuses on the strength guaranteed by a protection technique once it is located. However, the invisibility of a protected asset, that is, its ability to blend with the other non-protected areas, can significantly impact the attack time, thus increasing the security of the application.

Neural networks seem particularly promising in this context due to their ability to detect very well-hidden patterns, even in complex and long collections of data. The current state-of-the-art works seem to favor two main techniques in this regard:

- NLP (Natural Language Processing) approaches: although NLP was born to deal with human languages (e.g., English, Spanish, or Italian), it can also be used to reason on programming statements and assembly instructions since they are languages too;
- CNNs (Convolutional Neural Networks): these deep neural networks are commonly used to analyze images and videos. However, due to their flexibility, they can also be used to perform various time-sequences analyses.

Kim et al. [10] proposed using a neural network to detect obfuscated functions in Intel binaries. Their work focuses on some protection techniques offered

by Obfuscator-LLVM¹. After disassembling the binary, the proposed approach involves counting the occurrence of some specific mnemonics (e.g., `add`, `mov`). This information is then given as input to a fully-connect neural work for classifying the code regions.

This approach showed good accuracy of 91% when a single obfuscation technique is used. However, when two protections are applied to the same code region, it drops to 85%.

In 2021, Canavese, Regano, and Basile [3] patented a methodology for identifying functions and assembly snippets protected with various protection tools on ARM and Intel architectures. Their approach uses state-of-the-art NLP neural networks and custom embeddings for the assembly instructions to locate obfuscated code regions. They experimentally showed that their system was able to detect functions protected with several well-known obfuscators, such as Obfuscator-LLVM, Tigress², and Diablo³. They reported that the accuracy for identifying a function obfuscated with a single protection technique is about 97% for both Intel and ARM architectures. In comparison, the accuracy slightly drops to 92% when the proposed system is used to pinpoint functions protected with two obfuscations.

In the same year, Jiang et al. [9] proposed a method based on LSTMs⁴ [8] to identify functions on Intel and ARM architecture protected with Obfuscator-LLVM. Their approach consists of three several features for each basic block⁵ and their adjacency matrix to model the control flow graph. This information is then given as input to an LSTM for the final classification of the functions. Their tests showed an accuracy of 95% for Intel applications and 99 % for Android binaries in detecting functions obfuscated with a single protection technique.

Zhao et al. [27] proposed a more complex approach based on a mix of CNNs and NLP techniques for identifying functions protected with Obfuscator-LLVM or Tigress on Intel architectures. Their procedure consisted of four steps:

1. disassembling the binary;
2. encoding the basic blocks using a custom embedding scheme;
3. performing a second encoding of the basic blocks embeddings with a CNN;
4. feeding an LSTM with the CNN output for the final classification of the function.

The tests showed that this approach could detect a function protected with a single protection with an accuracy of 91% and functions safeguarded by two protections with an average accuracy of 88%.

¹ <https://github.com/obfuscator-llvm/obfuscator>.

² <https://tigress.wtf/>.

³ <https://github.com/csl-ugent/diablo>.

⁴ LSTMs (Long-Short Term Memory) are a family of neural networks used to investigate time-sequences. They represent one of the modern key technologies to perform NLP, and several commercial products utilize them (e.g., Google Translator, Alexa).

⁵ A basic block is a sequence of instructions with exactly one execution flow ingress point (the first instruction) and one egress point (the last instruction).

2.2 Malware Analysis

Another important application in automatically detecting protected code regions is to perform malware analysis. Modern viruses, worms, and ransomware frequently obfuscate parts of their code to fool anti-viruses and anti-malware products. A common obfuscation technique used by malware is *packing*, which compresses the executable code. For instance, UPX⁶ (Ultimate Packer for eXecutables) is a well-known packer, also used by many viruses. When an application wants to execute its code, it unpacks (decompress) the code into memory and then launches it. While this kind of malware can evade static analysis, it cannot fool dynamic analysis since the executable code must be put into the memory to be run. Static analysis, however, is safer since it does not require running some potentially malicious code. The research efforts are particularly active in this context, especially in detecting malware on mobile devices such as Android smartphones and tablets.

Tang et al. [21] proposed an interesting approach based on a mix of classic NLP techniques and CNNs to identify obfuscated malware on Android systems. The first step in their process is to analyze the opcodes of the function under scrutiny and extract several features based on the TF-IDF (Term Frequency-Inverse Document Frequency) [17] technique. This algorithm computes various frequency-based metrics of the opcodes in a binary and creates a numerical vector for each opcode. These numbers are then combined to form a sort of grayscale image fed to a CNN for the classification of the function. They tested the effectiveness of their approach on a variety of Android malware applications obfuscated with the AVPASS tool⁷. They proved experimentally that their accuracy was respectively 96% and 95% for detecting the unobfuscated and obfuscated malware.

BLADE [19] is a tool proposed in 2021 by Sihag et al. able to detect obfuscated malware by analyzing the Dalvik bytecode of the Android OS. The key idea of this tool is to transform the malware classification problem into a document classification problem. The first phase in the authors' approach is to simplify the bytecode by grouping similar instructions into several categories. Then, several metrics (e.g., occurrence counts) are computed to transform the simplified bytecode into a numerical vector. This vector is then fed into a traditional machine-learning classifier to detect the presence of some malware. The authors tested various classic machine-learning models, such as random forests and k -NN, on multiple data sets containing a variety of obfuscated malware and experimentally found that their accuracy was about 96%.

Zhang et al. [26] compiled an interesting survey on various obfuscation detection (and deobfuscation) tools and techniques for forensic investigations of Android devices, focusing on malware analysis. For obfuscation detection, they examined eight publicly available tools. Most of the investigated tools use a similar approach: computing some numeric feature on the bytecode (e.g., counting

⁶ <https://upx.github.io/>.

⁷ <https://github.com/sslabs-gatech/avpass>.

the number of specific instructions) and then using some (machine-learning-based or not) classifier to detect the presence of some protected code automatically. They concluded that although several research papers are available in this area, most of the tools are not publicly available, thus prohibiting the improvement of their detection capabilities.

3 Code Understanding

After identifying an interesting area of code, potentially an asset, the attacker must first be able to understand the code's inner workings before circumventing the protection applied to the target code successfully. To understand the code, the attacker must first obtain an intelligible representation of it. This is typically achieved by disassembling or decompiling the code, getting respectively the assembly code (in the format of the binary's target machine, e.g., x86-64 or ARM), or a reconstructed version of the source code (with varying degrees of accuracy w.r.t. the original source code). Many disassemblers are available, including commercial solutions (e.g., IDA Pro, Binary Ninja, OBJ2ASM) and open-source ones (e.g., Capstone, objdump, gdb). Indeed, this task may take a non-negligible time, proportional to the skills and experience of the attacker and the tools at his disposal. Similarly, various decompilers are available, both commercial (e.g., Hex-Rays Decompiler) and free (e.g., Ghidra, RetDec). After obtaining an understandable representation of the code, the attacker may comprehend the target code statically or dynamically. In the first case, the attacker does not execute the code and tries to understand it, either by reading it directly or leveraging more user-friendly representations of it (e.g., Control Flow Graph, Data Dependency Graph). In the second case, the attacker observes the program at run-time, typically executing the target application with a debugger attached. Typically, dynamic code analysis is more straightforward and faster w.r.t. static analysis. However, it should be noted that protections able to prevent debugging are available [1]. Thus, the second option is not always viable.

Symbolic execution is a static program analysis technique that may be used to speed up the process of code understanding. This technique resorts to a mathematical representation of the code, typically modeling the latter via a graph-like state model. Using this mathematical representation, it is possible to prove mathematically the values that, given in input to the target application, will result in the execution of the target code. Furthermore, this will result in the list of instructions that will be executed given a specific set of inputs. In this way, it is possible to emulate the execution of the application without actually running it, thus circumventing anti-debugging protection. The idea in itself is not new; however, it has been gaining traction in the last years due to the increased available computational power in commodity hardware (thus enabling the more complex programs to be emulated). Also, an interesting application of this technique is the possibility of analyzing malware without resorting to sandboxed execution since many malicious applications can understand if they are executed in a sandboxed environment and consequently alter their run-time behavior to confuse security experts analyzing them.

One of the main problems of symbolic execution is path explosion since there is an exponential relation between the SLOC of the target application and the number of possible execution paths. This may render this technique unfeasible for large programs, notwithstanding the aforementioned increase in computational power in recent years. Concolic (i.e., Concrete-symbolic) execution tries to resolve this limitation by mixing static and dynamic code analysis. In particular, symbolic execution is still used for the main parts of the application that must be analyzed. However, other code areas are executed (i.e., concretely), driving their input with the results of symbolic execution to avoid their emulation. In this way, path explosion can be limited. Typical targets for run-time execution are standard libraries and system calls since they can be executed in isolation. This technique may still be used for malware analysis and on programs protected with anti-debugging techniques. Overall, this technique permits us to obtain in a faster way the same results of symbolic execution.

The works presented in the remainder of the section show applications of symbolic and concolic execution to two relevant cybersecurity problems: automated identification of vulnerabilities leading to information leakage and malware analysis. Indeed, these works prove that symbolic and concolic execution may be successfully used to automate tasks usually carried on by cybersecurity analysts, reducing the space for human error and increasing the efficiency of both vulnerability and malware detection. Future advancements in both these techniques may lead to an even greater amount of automation, thus leading to the widespread adoption of these approaches in commercial solutions.

3.1 Automated Identification of Information Leakage Vulnerabilities

A recent application of symbolic/concolic execution in this context is the automated identification of vulnerabilities leading to secret leakage during execution. A well-known example of such vulnerabilities is the Spectre family of attacks. These vulnerabilities exploit the speculative execution systems used by Intel CPUs to increase program execution speed. In particular, when a branch execution must be executed, the branch condition should usually be evaluated to select the branch where execution should continue. However, with speculative execution, a CPU component, named branch predictor, will try to guess the branch that must be executed, continuing program flow without actually evaluating the condition, which is postponed to increase efficiency. If the guess is deemed correct, the program execution usually continues. Instead, if the guess is incorrect, the program state is reversed, and the instructions of the right branch are executed. Unfortunately, wrong branch guesses may lead to side effects (e.g., on the contents of CPU caches) that attackers may leverage to recover application secrets. Applications of this attack family to recover cryptographic secrets have been demonstrated⁸.

Daniel, Bardin, and Rezk [5] adapted binary symbolic execution to consider the effects of speculative execution. They employed pruning techniques

⁸ <https://spectreattack.com/spectre.pdf>.

of the code graph representation used by symbolic execution, crafted for specific attacks in the Spectre family, to keep complexity at bay. They developed a Haunted RelSE tool to demonstrate their approach’s effectiveness. They successfully identified Spectre vulnerabilities in various cryptographic libraries, including OpenSSL. Furthermore, their tests lead to an interesting finding: defense techniques developed to mitigate Spectre vulnerabilities may introduce other vulnerabilities of the same family in the protected code.

Guo et al. [24] developed a plugin for KLEE, a concolic execution framework based on the LLVM compiler. This plugin, called SpecuSym, can detect cache side effects that may lead to information leaks by programmatically exploring all program states. They developed an ad-hoc model of speculative execution to assist concolic execution in taking into account the cache side effects during path exploration. Similarly, Wan et al. proposed KLEESpectre [15], another plugin for KLEE based on similar ideas of SpecuSym, but specifically developed to identify vulnerabilities of the Spectre family.

Borzacchiello, Coppa, and Demetrescu [2] introduced FUZZOLIC, a concolic analysis framework. In particular, they combine concolic execution with fuzzing. In this dynamic analysis technique, the application is executed many times with varying input, to detect bugs by observing if a particular set of inputs lead to unexpected run-time behaviors or program crashes. Since FUZZOLIC runs under QEMU, a well-known software emulation solution, it supports binaries developed for many CPU architectures. They have found various software bugs by combining such techniques and applying them to a set of Linux command-line tools, including memory leakage vulnerabilities.

3.2 Malware Analysis with Symbolic and Concolic Execution

As previously stated, one interesting application of symbolic and concolic execution is the possibility of securely analyzing malware without actually executing it, without resorting to sandboxed approaches that may be hampered by the sandbox detection included in advanced malware.

Sebastio et al. [18] introduced a framework for symbolic/concolic analysis of obfuscated malware. In particular, the authors leverage angr⁹, a binary analysis tool, to analyze the binary. The tool parameters are carefully selected to speed up the code analysis. Then, malware system calls are identified to populate an SCDG (System Call Dependency Graph) structure. This permits an accurate representation of the run-time behavior of the malware. Using gSpan7, a graph mining solution, the analyzed malware can be classified in one of the various families (e.g., , worm, ransomware, crypto-miner), with a 97% classification accuracy.

Van Ouytsel et al. proposed SEMA, a symbolic execution toolchain specialized in the detection of polymorphic malware, i.e., malware that can hinder classical static analysis solutions by dynamically changing the virus code at each execution while preserving its business logic. SEMA leverages an ad-hoc angr

⁹ <https://angr.io/>.

extension to obtain from the target malware the SCDG, with optimizations for frequent API calls to hasten the analysis. Then, classification of the malware may be performed through two different approaches, one based on graph mining and the other on deep learning. An interesting addition is the adoption of federated learning, where multiple devices may remotely collaborate to train and update the malware detection classifier base on deep learning.

Namani and Khan [14] leverage concolic execution to analyze malware, again for identification and classification purposes. The analysis is performed by first disassembling the malware to obtain both the binary header and calls to external libraries. A subsequent concolic execution is targeted on the code areas containing API calls. This results in a feature vector that can be fed to a machine-learning classifier. In particular, the author tested three different machine learning approaches (decision trees, random forests, and fully connected neural networks), with a resulting accuracy ranging from 92% to 97%.

Park et al. [15] introduced BDHunter, an automated system for the identification of malware behavior dispatchers, i.e., groups of branch instructions that lead to the execution of malicious actions. The system first identifies candidate behavior dispatchers with two different approaches based on identifying patterns in the target binary control flow graph and weighted API calls. The candidates are confirmed or disproven by analyzing run-time behavior via concolic execution. Indeed, malware typically checks various run-time conditions' inner workings before receiving commands from a C2 server.

4 Protection Removal or Bypass

After locating promising code areas in the target application and understanding the program's internal workings, the attacker may need to remove or bypass protections applied to the target code areas to be able to perform attacks ultimately. This step is typically done manually, editing or rewriting the binary code directly or using ad-hoc scripts to bypass a specific protection technique. Thus, only a sufficiently motivated and expert attacker can typically remove non-trivial software protections. However, the latest research in the field has been directed at automating this complex and time-consuming process, leveraging the possibilities given by the new findings in the machine learning and AI area. The work has been focused on obfuscation since it is the most common family technique and is typically employed by malware to escape classical identification mechanisms implemented by anti-malware solutions.

Menguy et al. [13] introduced a framework for deobfuscation based on a black-box approach called Xyntia. In particular, this framework performs deobfuscation by defining an optimization problem to generate a code that preserves the business logic of the target code while minimizing its complexity. The optimization problem solution is carried on with a variation of a well-known heuristic, ILS (Iterated Local Search). The authors provided experimental proofs of successful reconstruction of code obfuscated with various techniques, including

opaque predicates, an obfuscation approach able to increase code size by inserting branches to add dead code to the target area, and Mixed-Boolean Arithmetic (MBA), which increases the complexity of arithmetic expressions.

David, Coniglio, and Ceccato [6] proposed an approach to thwart various obfuscation techniques such as data encoding, MBA, and virtualization. The latter is a protection technique that translates the target code using a custom bytecode, which is interpreted at run-time by a VM included in the distributed protected application. The proposed tool performs deobfuscation in two phases, the first resorting to concolic execution to analyze the program and extract an abstract syntax tree representation of the target application and the second performing a top-down breadth-width search of the AST to obtain a deobfuscated version of the protected code.

A systematic literature review by Kochberger et al. [11] enumerates different solutions to thwart virtualization obfuscation. In their experiment, they deobfuscated various applications previously protected with virtualization, using 15 automatic deobfuscators, to assess their performance in understanding the virtualized bytecode. Furthermore, they organized deobfuscation techniques in state of the art into a taxonomy based on the analysis type performed by the deobfuscators (e.g., static, dynamic, or hybrid), the artifacts used to perform such analysis (e.g., traces, control flow graphs), and the level of automation achieved (fully or partially automated).

Suk, Bi, and Lee [20] developed SCORE, a tool for reversing Control Flow Flattening, one of the most used code obfuscation techniques. The tool starts by executing the code to recover information needed to perform the actual deobfuscation, such as the order of execution of instructions of obfuscated code areas and patterns of execution of program functions. Deobfuscation is then performed with a three-step process. First, the instructions are rearranged in the order of execution. Then, dead code elimination is achieved, given the prior results of code execution. Finally, various source code optimization techniques are performed to improve the readability of the obtained deobfuscated code.

You et al. [25] performed a study of two different code optimization and deobfuscation tools for Android applications, ReDex¹⁰ and Deguard¹¹. In particular, they generated a set of obfuscated Android applications using two different obfuscation tools (R8 and Obfuscapk) on three vanilla Android applications. They then performed deobfuscation on the applications using an ad-hoc tool for Android apps and compared the resulting source codes with the original unobfuscated ones to evaluate deobfuscation accuracy.

The works presented show that it is indeed possible to automate the task of reverse engineers since automatic deobfuscators have been successfully developed. Since code obfuscation is the most common technique employed to protect software, these advancements will certainly boost the research in the field, possibly leading to the development of completely new protection techniques and

¹⁰ <https://fbredex.com>.

¹¹ <https://apk-deguard.com>.

improvements of existing ob techniques to improve their resilience against automatic deobfuscation.

5 Conclusions

Reverse engineering a software application is inherently challenging and requires expertise, time, and patience. Even more so if the code to analyze is protected. Attackers typically crack software applications for illegal purposes, such as removing or bypassing license checks in commercial software and video games. However, software houses also frequently use reverse engineering techniques for defensive purposes, such as assessing the security of a protected application before its release or helping anti-viruses detect new and more advanced forms of malware.

Even in our technocentric world, this task is performed primarily manually via tools such as debuggers, decompilers, and disassemblers with minimal automated support. In the last few years, however, the surfacing of new techniques has started to change this trend.

In our analysis, two families of techniques will dominate the scenes in the reverse engineering field: machine learning techniques based on neural networks and symbolic/concolic execution approaches. None of these techniques is new: in fact, neural networks were born in the 1950s, and symbolic execution has been known since the 1970s. However, recent advancements in computation power have unleashed the ability to use more complex and demanding analyses that were prohibitive before.

Successfully automatizing the reverse engineering process will greatly impact the future of the software protection world in many different ways. First, it is foreseeable that companies will start using AI and symbolic execution-based techniques to aid their expert in releasing more secure and protected commercial software (and their patches) while significantly reducing their time-to-market window. On the other hand, anti-viruses and anti-malware programs will also benefit from these advancements. Machine-learning bases analysis can drastically enhance the detection of new breeds of malicious and obfuscated malware.

Acknowledgments. This work has been partly supported by the CyberSec4Europe project (Horizon 2020 proposal no. 830929).

References

1. Abrath, B., Coppens, B., Volckaert, S., Wijnant, J., Bjorn, S.D.: Tightly-coupled self-debugging software protection. In: Proceedings of SSPREW 2016: Workshop on Software Security, Protection, and Reverse Engineering. Los Angeles (USA), pp. 1–10 (2016). <https://doi.org/10.1145/3015135.3015142>
2. Borzacchiello, L., Coppa, E., Demetrescu, C.: Fuzzolic: mixing fuzzing and concolic execution. *Comput. Secur.* **108** (2021). <https://doi.org/10.1016/j.cose.2021.102368>

3. Canavese, D., Regano, L., Basile, C.: Method for the identification of protected assets in software binaries (2021). <https://www.knowledgeshare.eu/en/patent/method-for-the-identification-of-protected-assets-in-software-binaries/>, application number 102021000012488
4. Ceccato, M., Di Penta, M., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empir. Softw. Eng.* **19**(4), 1040–1074 (2013). <https://doi.org/10.1007/s10664-013-9248-x>
5. Daniel, L.A., Bardin, S., Rezk, T.: Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse. In: *Proceedings of NDSS 2021: Network and Distributed System Security Symposium*, pp. 1–18. Virtual conference (2021). <https://doi.org/10.14722/ndss.2021.24286>
6. David, R., Coniglio, L., Ceccato, M.: Qsynth - a program synthesis based approach for binary code deobfuscation. In: *Proceedings of BAR 2020: Workshop on Binary Analysis Research*. San Diego (USA), pp. 1–12 (2020). <https://doi.org/10.14722/bar.2020.23009>
7. Hänsch, N., Schankin, A., Protsenko, M., Freiling, F., Benenson, Z.: Programming experience might not help in comprehending obfuscated source code efficiently. In: *Proceedings of SOUPS 2018: USENIX Conference on Usable Privacy and Security*. Baltimore (USA), pp. 341–356 (2018)
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural Comput.* **9**, 1735–80 (1997). <https://doi.org/10.1162/neco.1997.9.8.1735>
9. Jiang, S., Hong, Y., Fu, C., Qian, Y., Han, L.: Function-level obfuscation detection method based on graph convolutional networks. *J. Inf. Secur. Appl.* **61**, 102953 (2021). <https://doi.org/10.1016/j.jisa.2021.102953>
10. Kim, J., Kang, S., Cho, E.-S., Paik, J.-Y.: LOM: lightweight classifier for obfuscation methods. In: Kim, H. (ed.) *WISA 2021*. LNCS, vol. 13009, pp. 3–15. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-89432-0_1
11. Kochberger, P., Schrittwieser, S., Schweighofer, S., Kieseberg, P., Weippl, E.: Sok: automatic deobfuscation of virtualization-protected applications. In: *Proceedings of ARES 2021: International Conference on Availability, Reliability and Security*. Benevento (Italy), pp. 1–15 (2021). <https://doi.org/10.1145/3465481.3465772>
12. László, T., Kiss, Á.: Obfuscating c++ programs via control flow flattening. *Annales Univ. Sci. Budapest* **30**, 3–19 (2009)
13. Menguy, G., Bardin, S., Bonichon, R., de Souza Lima, C.: AI-based blackbox code deobfuscation: understand, improve and mitigate. *CoRR* **abs/2102.04805** (2021). <https://arxiv.org/abs/2102.04805>
14. Namani, N., Khan, A.: Symbolic execution based feature extraction for detection of malware. In: *Proceedings of ICCCS 2020: International Conference on Computing, Communication and Security*, pp. 1–6. Virtual Conference (2020). <https://doi.org/10.1109/ICCCS49678.2020.9277493>
15. Park, K., et al.: Identifying behavior dispatchers for malware analysis. In: *Proceedings of ASIACCS 2021: Asia Conference on Computer and Communications Security*. Hong Kong (China), pp. 759–773 (2021). <https://doi.org/10.1145/3433210.3457894>
16. Regano, L., Canavese, D., Basile, C., Liroy, A.: Towards optimally hiding protected assets in software applications. In: *Proceedings of QRS 2017: International Conference on Software Quality, Reliability and Security*. IEEE, Prague (Czech Republic), pp. 374–385 (2017). <https://doi.org/10.1109/QRS.2017.47>
17. Salton, G., McGill, M.: *Introduction to Modern Information Retrieval*. McGraw-Hill (1983)

18. Sebastio, S., et al.: Optimizing symbolic execution for malware behavior classification. *Comput. Secur.* **93**, 101775 (2020). <https://doi.org/10.1016/j.cose.2020.101775>
19. Sihag, V., Vardhan, M., Singh, P.: Blade: robust malware detection against obfuscation in android. *Forensic Sci. Int. Digit. Investig.* **38**, 301176 (2021). <https://doi.org/10.1016/j.fsidi.2021.301176>
20. Suk, J.H., Lee, Y.B., Lee, D.H.: Score: source code optimization & reconstruction. *IEEE Access* **8** (2020). <https://doi.org/10.1109/ACCESS.2020.3008905>
21. Tang, J., Li, R., Jiang, Y., Gu, X., Li, Y.: Android malware obfuscation variants detection method based on multi-granularity opcode features. *Futur. Gener. Comput. Syst.* **129**, 141–151 (2022). <https://doi.org/10.1016/j.future.2021.11.005>
22. Viticchié, A., Regano, L., Basile, C., Torchiano, M., Ceccato, M., Tonella, P.: Empirical assessment of the effort needed to attack programs protected with client/server code splitting. *Empir. Softw. Eng.* **25**(1), 1–48 (2019). <https://doi.org/10.1007/s10664-019-09738-1>
23. Viticchié, A., et al.: Assessment of source code obfuscation techniques. In: *Proceedings of SCAM 2016: International Working Conference on Source Code Analysis and Manipulation*, pp. 11–20. IEEE, Raleigh (USA) (2016). <https://doi.org/10.1109/SCAM.2016.17>
24. Wang, G., Chattopadhyay, S., Biswas, A.K., Mitra, T., Roychoudhury, A.: Kleespectre: detecting information leakage through speculative cache attacks via symbolic execution. *Trans. Softw. Eng. Methodol.* **29** (2020). <https://doi.org/10.1145/3385897>
25. You, G., Kim, G., Je Cho, S., Han, H.: A comparative study on optimization, obfuscation, and deobfuscation tools in android. *J. Internet Serv. Inf. Secur.* **11** (2021)
26. Zhang, X., Breiting, F., Luechinger, E., O’Shaughnessy, S.: Android application forensics: a survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Sci. Int. Digit. Investig.* **39**, 301285 (2021). <https://doi.org/10.1016/j.fsidi.2021.301285>
27. Zhao, Y., et al.: Semantics-aware obfuscation scheme prediction for binary. *Comput. Secur.* **99**, 102072 (2020). <https://doi.org/10.1016/j.cose.2020.102072>