An Overview of the Endless Battle between Virus Writers and Detectors: How Compilers Can Be Used as an Evasion Technique

Michele Ianni[1], Elio Masciari[2] and Domenico Saccà[1]
[1]*DIMES, UNICAL, Via P. Bucci, Rende, Italy*
[2]*ICAR, CNR, Via P. Bucci, Rende, Italy*

Keywords: Metamorphic Malware, Obfuscation, Malware Detection, Single Instruction Compilers.

Abstract: The increasing complexity of new malware and the constant refinement of detection mechanisms are driving malware writers to rethink the malware development process. In this respect, compilers play a key role and can be used to implement evasion techniques able to defeat even the new generation of detection algorithms. In this paper we provide an overview of the endless battle between malware writers and detectors and we discuss some considerations on the benefits of using high level languages and even exotic compilers (e.g. single instruction compilers) in the process of writing malicious code.

1 INTRODUCTION

History of malware, short for malicious software, is characterized by the endless battle between malware writers and detectors. Since detection strategies are becoming more and more complex, malware writers have to invent new techniques in order to evade detection. Today we can find many viruses that can be considered as pieces of art because they employ several clever ideas in order to keep themselves as stealth as possible. The increasing complexity of new malware is posing new intriguing challenges both from the malware writer perspective and detection mechanisms. In order to implement complex malware, able to spread itself on various operating systems and architectures, it could be useful to move from pure assembly implementations, to malware written using high level languages. In this respect the use of compilers is a key concept to take into account. Compilers, in fact, can be used to implement metamorphic techniques and obfuscation and can build executables able to defeat new detection mechanisms based on the extraction of semantic patterns from the binaries. The paper is organized as follows: in section 2 we describe several techniques used by malware writers in order to avoid detection. In section 3 we show some of the strategies used by antivirus software to detect malicious code and we focus our attention on CFG based detection. In section 4 we discuss the benefits related to the use of compilers in the process of writing malicious code and we show the advantages of using single instruction compilers as an evasion technique. Finally in section 5 we draw our conclusions.

2 EVASION TECHNIQUES

The most widespread technique used by commercial anti-malware systems in order to detect viruses is based on malware signatures (Idika and Mathur, 2007). They are invariant patterns, usually taken from the program's code or raw file content, used to uniquely identify the given malware. To evade signature based scanners many today viruses (called metamorphic) are able to transform themselves during the propagation phase, without losing their capabilities (You and Yim, 2010). To achieve this goal several metamorphic transformation are used, including code permutation, garbage code insertion, code shrinking and expansion, register renaming, encryption (Collberg et al., 1997; Barak et al., 2001; Beaucamps and Filiol, 2007). The result of these transformations is a brand new virus that, while keeping the functionality of its predecessor, present a different structure and then a different signature, thus evading detection (You and Yim, 2010). As explained in (Bonfante et al., 2007), we define $M \subset P$ to be the set of malicious programs, where P is the set of all programs and $\mathbb{S}$ to be the set of signatures. A detector is then a function $D : P \times M \to \{0, 1\}$. A program p is detected if there is a signature $m \in \mathbb{S}$ such that $D(p, m) = 1$. In the case

Ianni, M., Masciari, E. and Saccà, D.
An Overview of the Endless Battle between Virus Writers and Detectors: How Compilers Can Be Used as an Evasion Technique.
DOI: 10.5220/0007922802030208
In *Proceedings of the 8th International Conference on Data Science, Technology and Applications (DATA 2019)*, pages 203-208
ISBN: 978-989-758-377-3

of malwares $D(p,m) = 1 \iff m$ is a pattern derived from p.

Using signature for malware detection is efficient only if it is applied to known malware. This technique, compared to dynamic analysis techniques, has less scanning time, lower false-positive ratio and doesn't suffer of the risks of system infection due to the execution of the malware. The use of code obfuscation techniques allows to easily generate variants of known malware resulting in new malware that cannot be detected by signature based scanners and are harder to comprehend for an human analyst. As stated in (Lyda and Hamrock, 2007) more than 80% of malware is packed, in accordance to (Stepan, 2006) almost 50% of new malware in 2006 were existing malware obfuscated with packing techniques. The same trend is linked to the use of other obfuscation approaches. There exist many examples of code obfuscation designed to avoid AV scanners detection (Driller, 2002; Mohanty, 2005; Rajaat, 1999; Julus, 2000) and all of them are able to easily evade signature based malware detectors.

As described in (You and Yim, 2010) except for packing, the first obfuscation technique historically used is encryption (Schiffman, a; Schiffman, b; Wong and Stamp, 2006). The executable body is crypted and the malware adds to it a decryptor that provides decryption of the body at program runtime. Since at every infection the cryptographic keys used are different the crypted virus body will be always different. One of the first viruses that developed this strategy is Cascade, followed by Win95/Mad and Win95 Zombie (Szor and Ferrie, 2001). Some of this kind of viruses use also multiple layers of encryption (Win32/Coke). The major limitation of this approach is that in most cases the decryption is in clear text and, since it is always the same, it can be used in order to generate a signature. To overcome this limitation malware writers created new malware able to change also decryptor code. This led to the birth of polymorphic malware (Rajaat, 1999), that, using many different techniques (Wong and Stamp, 2006; Christodorescu and Jha, 2006; Konstantinou and Wolthusen, 2008) are able to generate always different decryptors, without invariant patterns that could be used as signatures. The first viruses that used a real 32-bit polymorphic engine were Win95/Marburg and Win95/HPS. They have been developed and spreaded online many polymorphic engines, among these we can cite *"The Mutation Engine"* (*MtE*) (Schiffman, a) that is able to easily convert non obfuscated code into polymorphic malware. Altough polymorphic malware are effective against signature based detection, they can be detected using more refined techniques. After the decryption phase, in fact, the body of the virus will be always the same. Using sandboxing techniques (Schiffman, a; Schiffman, b) the detectors are able to emulate malware in a controlled environment, allowing the decryptor to decrypt malware body in memory. At this point it is still possible to use signature based techniques on the decrypted body. To prevent malware emulation several armoring techniques have been proposed (Schiffman, a) but the improvements in sandboxing mechanisms brought many of them to be ineffective. To overcome all these limitations malware writers brought obfuscation to a new level: metamorphic malware (Driller, 2002; Julus, 2000) (Schiffman, b; Wong and Stamp, 2006; Christodorescu and Jha, 2006; Konstantinou and Wolthusen, 2008). They are malware able to transform their own body during infection phase. At each iteration metamorphic malware is rewritten so that each succeeding version of the code is different from the preceding one.

There are numerous techniques used by this kind of malware (and often by polymorphic malware too):

- *Register swapping*: is a technique used, for example, by Vecna's Win95/Regswap virus (Wong and Stamp, 2006). It consists in changing registers used in various instructions during the evolution from generation to generation. Wildcard searching makes this technique ineffective.
- *Instruction substitution*: it is based on substituting single, or groups, of instructions with other instructions (or groups of them). The new instructions will be equivalent to the previous ones in functionalities but syntactically different (Konstantinou and Wolthusen, 2008).
- *Garbage instructions insertion*: is a technique based on the insertion of garbage instructions, that are useless for the execution of the program. Their only goal is to vary malware body (Balakrishnan and Schulze, 2005; Wong and Stamp, 2006; Konstantinou and Wolthusen, 2008). They can be single instructions or sequences that perform useless operations leaving unaltered the state of the program or even instructions located in areas of the program that will never be executed. In this case we are talking about *dead-code*.
- *Transposition*: this name is used to define many instruction reordering techniques that leave unaltered the flow of the program (Christodorescu and Jha, 2006). One of these technique consist in randomly reordering some instructions then using unconditional jumps to reconstruct the original flow. In a more elegant way it is possible to isolate independent groups of instructions then modify their order. In this case, since the sequences are

independent, there is no need to make use of unconditional jumps. Finding independent groups of instructions is not an easy task to perform, so, often developers use easier techniques, which can be considered a variant of the first one. It is based on the reordering of the various subroutines that are present inside the malware (Christodorescu and Jha, 2006). One of the malware that used this approach is Win32/Ghost. If we have *n* subroutines we are able to generate up to *n*! different variants.

- *Code integration*: is the most sophisticated technique used to obfuscate code. It has been introduced by the virus writer Zombie in Win95/Zmist (Zombie Mistfall). It consists in decompiling the program to infect in distinct parts and then inserting malware code between them. At the end the original program and the malware are reassembled in a single executable.

3 MALWARE DETECTION

Although several theoretical studies (Cohen, 1987; Chess and White, 2000) have proved that an algorithm able to detect all types of malware can not exist, a lot of effort has been put to improve detection mechanisms. Several methods have been proposed, varying from support vector machine (SVM) (Burges, 1998), to decision trees (Kolter and Maloof, 2004; Moser et al., 2007), to Naive Bayes Method (Schultz et al., 2001). There are two different approaches in malware detection: *static* and *dynamic analysis*. The first analyzes a binary without executing it. Since this technique is safer, faster and easier to implement than dynamic analysis, it is the most widespread approach, even if it is more limited than the latter. Some examples of operations that are performed by static analysis are finding patterns on executables and code flow analysis. In addition to be fast and safe, static analysis has the advantage of reaching complete application code coverage, thus reducing the number of false positives in malware detection. The main problem in using static analysis is that it is very difficult to detect unknown malware. Dynamic analysis, instead, runs malware in a sandbox simulating the behaviour of a real environment, monitoring all system calls. It is clear that the speed of performing this kind of analysis is much slower than static analysis techniques. In addition to that, several techniques have been proposed from malware writers in order to check if the infected executable is running inside a virtual machine, and, in that case, changing its own behaviour in a non offensive one. To overcome the limitation of static analysis they have been introduced many techniques based on the concept of *code normalization*. These techniques try to reduce obfuscated code to a base form that is the same for all obfuscated variants of the same executable. Many different approaches to code normalization have been proposed. Among them we can cite Christodorescu et al. (Christodorescu et al., 2005), Walenstein et al. (Walenstein et al., 2006) and Lakhotia et al. (Lakhotia and Mohammed, 2004). In the latter the base form is called “zero form” and the process to reduce an executable to that form is called “zeroing”. As herm1t states in (herm1t, 2002) if this kind of approach was perfect a tool that could perform zeroing reducing all possible variants of a virus into a single “normalized” form (not necessarily optimal), could then use this strategy with every algorithm thus proving or refuting their identity. That’s known to be indecidable. The effort of the creators of detectors is then focused on capturing semantic pattern inside an executable rather than invariant synctactic features. These techniques vary from API call analysis (Sathyanarayan et al., 2008) to Control Flow Graph analysis.

As explained in (Shoshitaishvili et al., 2016) a *Control Flow Graph* (*CFG*) is a graph in which the nodes are basic blocks of execution and the edges are possible control flow transfers between them. They are used both for malware detection and vulnerability discovery. CFG recovery is a widely discussed topic in literature, we can cite (Cifuentes and Van Emmerik, 2001; Kinder and Veith, 2008; Kruegel et al., 2004; Schwarz et al., 2002; Troger and Cifuentes, 2002; Xu et al., 2009). The general method to recover a CFG (for more details see (Shoshitaishvili et al., 2016)) consist on using a recursive algorithm like 1:

Algorithm 1: Control Flow Graph recovery.

```
 1: function CFG_RECOVERY
 2:     $basic\_block\_queue \leftarrow \{\}$
 3:     $CFG \leftarrow \{\}$
 4:     $basic\_block\_queue.push(B_0)$   ▷ $B_0$ is the first basic block
 5:     $CFG.add(B_0)$
 6:     while basic_block_list is not empty do
 7:         $\overline{B} \leftarrow basic\_block\_queue.pop()$
 8:         for each basic block $B_i$ that can follow $\overline{B}$ in the execution do
 9:             if $CFG$ does not contain $B_i$ then
10:                 $CFG.add(B_i)$
11:                 $basic\_block\_queue.push(B_i)$
12:             end if
13:             Connect $\overline{B}$ to $B_i$
14:         end for
15:     end while
16: end function
```

Of course there exist more refined ways to recover a CFG.

Many CFG recovery algorithms deal with the problem of indirect jumps. An indirect jump occurs when the control flow is transferred to a target represented by a value in a register or to a memory location. This makes flow analysis much harder because the destination of the jump is not easily resolvable. This is because it could depend from computations specified in code, from the application context or even from function pointers used in object oriented languages to implement object polymorphism (Shoshitaishvili et al., 2016).

3.1 Control Flow Graph based Malware Detection

As previously stated, due to the difficulty on isolating invariant synctactic features of a self-mutating malware, the effort of the creators of detectors is focused on capturing semantic patterns. CFG are widely used to find similiarities among executables (Dullien and Rolles, 2005) and, more in detail, among malwares (Cesare and Xiang, 2010a; Briones and Gomez, 2008; Bonfante et al., 2007; Eskandari and Hashemi, 2011; Cesare and Xiang, 2010b; Jeong and Lee, 2008; Lee et al., 2010; Bruschi et al., 2006). The techniques proposed in literature are based on generating the CFG of a program P to analyze. The generated CFG is the compared to a set of CFGs of known viruses in order to find isomorphic components. Usually ((Bruschi et al., 2006)) before extracting the CFG from a program P a set of normalization operations (Lakhotia and Mohammed, 2004) are performed on the binary. This step aims to reduce the effects of mutation techniques. The normalized binary is then used to extract the CFG which is then compared to CFG extracted by normalized known malware. If the CFG of the normalized program contain a subgraph isomorphic to the CFG of a malware, then the executable is marked as malicious.

4 USING COMPILERS TO EVADE DETECTION

The advances in malware detection and the plethora of different devices and operating systems in use nowadays, pose new intriguing challenges to malware writers. The use of assembly language is becoming more and more painful, because of the difficulties involved to write portable and easy-to-support code. In the forward-looking article “Recompiling the metamorphism” (herm1t, 2002), the author, herm1t, suggests to make use of high level languages in order to overcome the difficulties involved in developing malware in pure assembly. He outlines how using a high level language gives to the developer the opportunity to easily extract additional information from the code, rather than builtin support for features like hashes, iterators or objects. The idea of ”recompiling the metamorphism” is without any doubt interesting and introduces many advantages to the virus writer, however, not all the benefits of using compilers have been considered in detail. In order to evade new anti-malware techniques based on the extraction of semantic patterns from executables, compilers could play an important role. They, in fact, are able to generate executables characterized by very different structures and could be used in order to defeat detection mechanisms. In this respect we take into accounts the benefits introduced by using exotic compilers, like the `M/o/Vfuscator2` [1] in order to obtain different CFGs, thus fooling the CFG based detection.

4.1 Single Instruction Compilers as an Evasion Technique

In (Dolan, 2013), Dolan demonstrates the Turing-completeness of the `x86` instruction `mov`. After the publication of the article many people started to implement, most of the time for fun, single instruction compilers, capable to compile arbitrary programs into lists of only `mov` instructions. Several different instruction turned out to be Turing-complete, so many different single instruction compilers arose [2]. Even if at first sight it may seem just like a funny fact, the use of single instruction compilers has very interesting consequences. Since in single instruction compiled programs comparisons, jumps, function calls are all implemented with a single instruction, the resulting CFG is a single (usually long) basic block. This result is very interesting, because having a CFG composed by a single basic block make ineffective all detection mechanisms based on CFG isomorphism detection. Using a single instruction compiler, however, has its drawbacks. First of all a program compiled with a single instruction compiler could be considered suspicious, thus marked as malicious. In addition to that, the size of a program compiled with a single instruction compiler, is, most of the time, much bigger than the size of the same program compiled using the entire set of instructions. This introduces some problems to virus writers that in many cases have a lim-

[1]https://github.com/xoreaxeaxeax/movfuscator

[2]https://github.com/xoreaxeaxeax/movfuscator/tree/master/post

ited space in the binary they are going to infect, so they should keep the malicious code as small as possible. Sometimes even the performances of the malicious code is important and programs compiled with a single instruction compiler usually are slower than traditional ones in terms of execution speed. These problems lead to rethink the way single instruction compilers are used for evasion purposes. A simple but effective solution could be splitting the source code at compilation time. In our implementation we mark with source code annotation the functions that we want to compile with a single instruction compiler, then at compilation time, making use of the tools provided by LLVM (Lattner and Adve, 2004) we are able to build an executable that uses the full set of instructions for the code that cannot be used to determine its malicious behaviour and a single instruction for the malware routines. In this way we are able to build a much smaller executable that is still able to fool CFG based malware detection. It is important to underline that the single block of single instruction compiled code can be furtherly modified using the metamorphic techniques described in section 2. This single block can be also manipulated in order to obtain different CFGs, this can be easily done by inserting jumps or comparisons, thus creating branches in the graph. To further variate the result of the obfuscation, several different single instruction compilers can be adopted, resulting in a great variety of CFGs, making very hard for the detectors to extract signatures, both syntactically, due to metamorphic transformation, and semantically, thanks to the always changing CFG.

5 CONCLUSIONS

In this paper we presented and overview of the techniques used by malware in order to avoid detection as well as some detection mechanisms. We showed the benefits related to the use of compilers on the process of malware creation and we proposed the use of single instruction compilers as an evasion mechanisms for CFG based malware detection. In order to overcome the limitations of this kind of compilers we proposed several solutions that can greatly increase the ability of malware to hide itself from detectors.

REFERENCES

Balakrishnan, A. and Schulze, C. (2005). Code obfuscation literature survey. *CS701 Construction of compilers*, 19.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2001). On the (im) possibility of obfuscating programs. In *Annual International Cryptology Conference*, pages 1–18. Springer.

Beaucamps, P. and Filiol, É. (2007). On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3(1):3–21.

Bonfante, G., Kaczmarek, M., and Marion, J.-Y. (2007). Control flow graphs as malware signatures. In *International workshop on the Theory of Computer Viruses*.

Briones, I. and Gomez, A. (2008). Graphs, entropy and grid computing: Automatic comparison of malware. *Virus Bulletin*, pages 1–12.

Bruschi, D., Martignoni, L., and Monga, M. (2006). Detecting self-mutating malware using control-flow graph matching. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 129–143. Springer.

Burges, C. J. (1998). A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167.

Cesare, S. and Xiang, Y. (2010a). Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*, pages 61–70. Australian Computer Society, Inc.

Cesare, S. and Xiang, Y. (2010b). A fast flowgraph based classification system for packed and polymorphic malware on the endhost. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 721–728. IEEE.

Chess, D. M. and White, S. R. (2000). An undetectable computer virus. In *Proceedings of Virus Bulletin Conference*, volume 5, pages 1–4.

Christodorescu, M. and Jha, S. (2006). Static analysis of executables to detect malicious patterns. Technical report, WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES.

Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., and Veith, H. (2005). Malware normalization. Technical report, University of Wisconsin.

Cifuentes, C. and Van Emmerik, M. (2001). Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40(2-3):171–188.

Cohen, F. (1987). Computer viruses. *Computers & security*, 6(1):22–35.

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

Dolan, S. (2013). mov is turing-complete. Technical report, Tech. rep. 2013 (cit. on p. 153).

Driller, M. (2002). Metamorphism in practice. *29A Magazine*, 1(6).

Dullien, T. and Rolles, R. (2005). Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3.

Eskandari, M. and Hashemi, S. (2011). Metamorphic malware detection using control flow graph mining. *Int. J. Comput. Sci. Network Secur*, 11(12):1–6.

herm1t (2002). Recompiling the metamorphism. https://83.133.184.251/virensimulation.org/lib/vhe11.html. Accessed: 2018-11-13.

Idika, N. and Mathur, A. P. (2007). A survey of malware detection techniques. *Purdue University*, 48.

Jeong, K. and Lee, H. (2008). Code graph for malware detection. In *2008 International Conference on Information Networking*, pages 1–5. IEEE.

Julus, L. (2000). Metamorphism. *29A Magazine*, 1(5).

Kinder, J. and Veith, H. (2008). Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*, pages 423–427. Springer.

Kolter, J. Z. and Maloof, M. A. (2004). Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM.

Konstantinou, E. and Wolthusen, S. (2008). Metamorphic virus: Analysis and detection. *Royal Holloway University of London*, 15:15.

Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. (2004). Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18.

Lakhotia, A. and Mohammed, M. (2004). Imposing order on program statements to assist anti-virus scanners. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 161–170. IEEE.

Lattner, C. and Adve, V. (2004). 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.

Lee, J., Jeong, K., and Lee, H. (2010). Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 1970–1977. ACM.

Lyda, R. and Hamrock, J. (2007). Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45.

Mohanty, D. (2005). Anti-virus evasion techniques and countermeasures. *Published online at http://www.hackingspirits. com/eth-hac/papers/whitepapers. asp.*, 18.

Moser, A., Kruegel, C., and Kirda, E. (2007). Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE.

Rajaat (1999). Polimorphism. *29A Magazine*, 1(3).

Sathyanarayan, V. S., Kohli, P., and Bruhadeshwar, B. (2008). Signature generation and detection of malware families. In *Australasian Conference on Information Security and Privacy*, pages 336–349. Springer.

Schiffman, M. A brief history of malware obfuscation: Part 1 of 2. Published online at https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2. Accessed: 2018-11-13.

Schiffman, M. A brief history of malware obfuscation: Part 2 of 2. Published online at https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_2_of_2. Accessed: 2018-11-13.

Schultz, M. G., Eskin, E., Zadok, F., and Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE.

Schwarz, B., Debray, S., and Andrews, G. (2002). Disassembly of executable code revisited. In *Reverse engineering, 2002. Proceedings. Ninth working conference on*, pages 45–54. IEEE.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al. (2016). Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE.

Stepan, A. (2006). Improving proactive detection of packed malware. *Virus Bulletin*, 1.

Szor, P. and Ferrie, P. (2001). Hunting for metamorphic. In *Virus bulletin conference*. Prague.

Troger, J. and Cifuentes, C. (2002). Analysis of virtual method invocation for binary translation. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 65–74. IEEE.

Walenstein, A., Mathur, R., Chouchane, M. R., and Lakhotia, A. (2006). Normalizing metamorphic malware using term rewriting. In *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop on*, pages 75–84. IEEE.

Wong, W. and Stamp, M. (2006). Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229.

Xu, L., Sun, F., and Su, Z. (2009). Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep.*

You, I. and Yim, K. (2010). Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297–300. IEEE.