MetaHunt: Towards Taming Malware Mutation via Studying the Evolution of Metamorphic Virus

Li Wang
lzw158@ist.psu.edu
The Pennsylvania State University
University Park, PA 16802, USA

Dongpeng Xu
dongpeng.xu@unh.edu
University of New Hampshire
Durham, NH 03824, USA

Jiang Ming
jiang.ming@uta.edu
University of Texas at Arlington
Arlington, TX 76019, USA

Yu Fu
yuf123@ist.psu.edu
The Pennsylvania State University
University Park, PA 16802, USA

Dinghao Wu
dwu@ist.psu.edu
The Pennsylvania State University
University Park, PA 16802, USA

ABSTRACT

As the underground industry of malware prospers, malware developers consistently attempt to camouflage malicious code and undermine malware detection with various obfuscation schemes. Among them, metamorphism is known to have the potential to defeat the popular signature-based malware detection. A metamorphic malware sample mutates its code during propagations so that each instance of the same family exhibits little resemblance to another variant. Especially with the development of compiler and binary rewriting techniques, metamorphic malware will become much easier to develop and outbreak eventually. To fully understand the metamorphic engine, the core part of the metamorphic malware, we attempt to systematically study the evolution of metamorphic malware over time. Unlike the previous work, we do not require any prior knowledge about the metamorphic engine in use. Instead, we perform trace-based semantic binary diffing to compare mutation code iteratively and memoize semantically equivalent basic blocks. We have developed a prototype, called *MetaHunt*, and evaluated it with 1,400 metamorphic malware variants. Our experimental results show that MetaHunt can accurately capture the semantics of unknown metamorphic engines, and all of the comparisons converge in a reasonable time. Besides, MetaHunt identifies several metamorphic engine bugs, which lead to a semantics-breaking transformation. We summarize our experience learned from our empirical study, hoping to stimulate designing mutation-aware solutions to defend this threat proactively.

CCS CONCEPTS

• **Security and privacy → Intrusion/anomaly detection and malware mitigation**.

SPRO'19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6835-3/19/11…$15.00
https://doi.org/10.1145/3338503.3357720

KEYWORDS

Malware detection, metamorphic virus, binary diffing, binary code semantics analysis

ACM Reference Format:
Li Wang, Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2019. MetaHunt: Towards Taming Malware Mutation via Studying the Evolution of Metamorphic Virus. In *3rd Software Protection Workshop (SPRO'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338503.3357720

1 INTRODUCTION

The malicious software (malware) underground market has evolved into a multi-billion dollar industry [6]. Driven by the rich profit, there has been consistent growth in the number and diversity of malware. According to a Panda Security Lab annual report [40], in 2017 alone, the total number of malware samples in circulation is as high as 75 million, 1.4 times the number of malware found in 2016. Relentless malware developers typically apply various obfuscation schemes (e.g., packer, polymorphism, and metamorphism) [37, 45] to camouflage arresting features, circumvent malware detection, and impede reverse engineering attempts. Among these obfuscation techniques, metamorphism is widely believed to be a panacea to thwart the signature-based malware scanning approaches [1, 47, 56], which are still by far the most widely used anti-malware solution in practice [8]. The core of metamorphic malware is a metamorphic engine (i.e., morphing engine). Each time a metamorphic malware sample executes or propagates, the metamorphic engine mutates the instructions that are loaded into memory by various methods such as register swapping, instruction substitution, instruction reordering, and junk code insertion. As a result, the old version is transformed into a syntactically different but semantically equivalent variant. In this way, a metamorphic malware sample becomes a moving target for analysis as the *archetype*[1] evolves from generation to generation. Consequently, the signature-based anti-malware approaches become insufficient to capture the numerous ostensibly different variants for a particular instance of malware, as illustrated in Figure 1. A striking example is from Leder et al.'s study [27] in

[1]The term "archetype" means the initial un-mutated version, from which the mutation starts.

2009. They report that only 12.6% of the files infected by the metamorphic malware Lexotan32 are detected by a total of 40 malware scanners in VirusTotal[2] and no single scanner can identify all the infected samples.

The prototype of metamorphic malware first emerges in the DOS days [48] and is constantly evolving on Windows platforms [3, 17]. However, compared with packer and polymorphism, metamorphism obfuscation was not widely adopted in the past. The major reason is that developing a full-fledged metamorphic engine is highly complicated, especially for self-propagating malware, which typically attaches the metamorphic engine in its code. For example, the relatively sophisticated sample, MetaPHOR (a.k.a.W32.Simile and W32.Etap) has about 14,000 lines of assembly code and more than 90% of its code is occupied by the metaphoric engine [19]. In recent years, with the advent of automated development toolkits, such as LLVM [26], SecondWrite [2], and Uroboros [51, 52], developing a powerful metamorphic engine will become relatively easy. For example, LLVM has been actively employed to facilitate malware mutation and diversification [24, 42, 49]. Therefore, we estimate that new malware variants with an advanced metamorphic engine will outbreak in the foreseeable future. To keep ahead in the malware defense arms race, we have to measure the risks of metamorphic malware and develop effective countermeasures.

A major challenge in metamorphic malware analysis is to design a general and automatic technique to capture all possible mutations [44]. Previous research work relies on studying the similarities before/after metamorphism and can be classified into three categories. The first category measures the similarity of static features such as control flow graph [4], opcode statistical signatures [10], instruction hidden Markov model [55], and characteristic value set [27]. Chouchane et al. [9] introduce an engine-specific scoring signature to match metamorphic engines. These approaches gear toward fast filtering out simple metamorphic malware, but they are brittle to defeat the sophisticated ones whose code are even encrypted [43]. Besides, the metamorphic engines can be decoupled from the malicious code to mutate non-propagating malware offline. For example, for the highly metamorphic malware created by NGVCK (Next Generation Virus Creation Kit) [55], their engines are separated from the malicious body. In that case, the "engine signature" approach is futile. The second category is based on the idea that the malicious behavior is not changed during code mutation. They detect metamorphic malware by measuring the similarity of API call sequences or graphs [28, 34, 56]. The main drawback to these API call approaches is that they regard the code mutation as a "black box", lacking an illuminating insight into the metamorphic engine.

The third category, also the most advanced one, aims at capturing the metamorphic engine's semantics [11]. They model the metamorphism either by semantic juice [25], algebraic specification [53], or abstract interpretation [13, 14]. The key design of metamorphic engine is a set of morphing rules (e.g., equivalent instruction substitution patterns), which guide how to transform instructions to their equivalent ones but with different syntax. A common assumption in the third category is that the metamorphic transformation rules are well known. Since several prototypes of metamorphic malware have been well studied or open sourced [19, 38], it seems that the prior knowledge about morphing rules can be collected easily. However, such optimistic assumption does not always hold in practice. It is always possible for an expert malware developer to design an alternative mutation way [39]. Unfortunately, manually tracing metamorphic mutations often cost several days or even weeks of tedious work, and the results are incomplete and error-prone as well.

In this work, we present *MetaHunt*, to study the evolution of metamorphic malware mutation over time. Our purpose is to understand the diversity of the metamorphic transformation comprehensively, and provide the insight of the mutation mechanism behind the metamorphic malware, which further helps stimulate the development of mutation insensitive malware protection solution. Different from the previous work, we do not assume the knowledge about the specific metamorphic engine in use. Instead, we study how a metamorphic engine mutates the code via *iteratively comparing input-related mutation code and memoizing equivalent basic blocks*. There are two key observations behind our approach. The first one is the metamorphic mutation is a semantics-preserving transformation. Therefore, ostensibly different code pairs but with the same function can be matched by state-of-the-art semantics-based binary diffing techniques [21, 30, 35]. The second one is, compared to other metamorphic transformation methods, the effect of equivalent instruction substitution is harder to reverse (e.g., via code normalization [5]) because of the cumbersome x86 instruction set architecture. Meanwhile, the sets of pure equivalent instruction substitution patterns are also limited [33, 50]. For example, the code substitution table of MetaPHOR consists of 94 alternative instruction sequences [19]. Consequently, after our preprocessing to remove some mutation methods such as junk code and opaque predicates, the iteration of comparing metamorphic mutations is not endless, which will converge when no new semantically equivalent code is discovered.

More specifically, given two metamorphic mutations, we first identify the basic blocks that can be affected by inputs via multi-tag taint analysis. Next, we perform normalization to reverse the mutation methods that may affect the scope of a basic block. After that, we represent the semantics of a basic block as a set of logical formulas by symbolic execution. Then we compare these logical formulas to find semantically equivalent basic block pairs with a theorem prover. After that, the semantically equivalent basic blocks are *memoized* in a union-find set [12], an efficient tree-based data structure. During successive comparisons, we continue to compare metamorphic variants and maintain the corresponding union-find sets until reaching a fixed point, that is, there is little or no increase in the size of the union-find sets. At that point, we call that we have explored *the metamorphic malware mutation evolution*. Although theoretically the attempt to find all the metamorphic mutations is equal to solving *the halting problem* [25], the collected information has many interesting implications from the practical point of view. For example, a mutation insensitive signature can be generated to capture all possible metamorphic variants; malware lineage information [23] can even be recovered as well.

[2]https://www.virustotal.com/

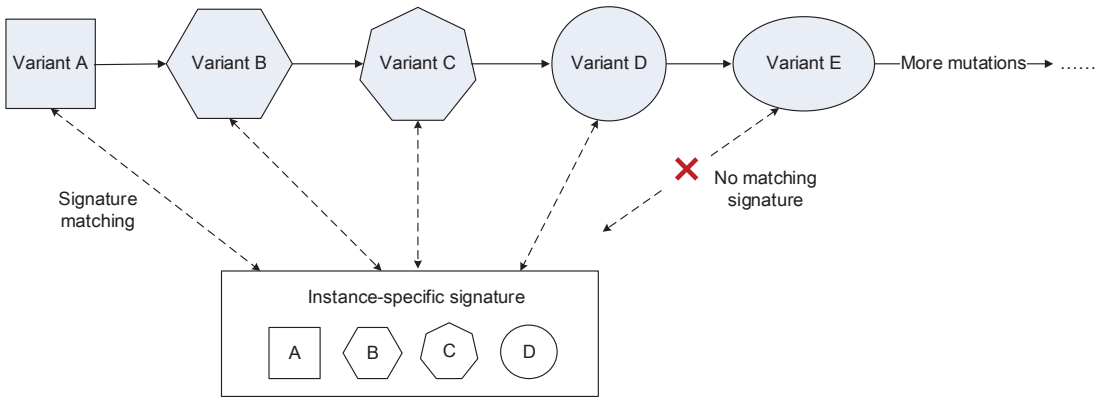


Figure 1: Metamorphic malware can evade conventional signature-based anti-malware solution.

We have implemented a prototype of MetaHunt on top of the BitBlaze [46] binary analysis platform. MetaHunt not only improves the semantics-based binary diffing technique in the resilience to highly obfuscated binary code, but also in the better performance. We perform a solid empirical study with 1,400 metamorphic malware samples, which are generated by nine metamorphic engines, including two advanced malware mutation tools based on LLVM [24, 41]. The evaluation shows that the iteration of comparing metamorphic malware variants converges in a reasonable time. Compared to manually reverse engineering of malware, MetaHunt's exploration result provides a comprehensive understanding about the mechanism of a metamorphic engine. In addition, MetaHunt identifies several buggy metamorphic engine implementations that ignore subtle side effects of the x86 instructions. Our MetaHunt prototype gives a method to record and compare the semantics of the metamorphic malware, which provides some feasible hints for the mutation insensitive anti-malware solutions. The result demonstrates that MetaHunt is an appealing complement to existing metamorphic malware defenses.

In summary, the contributions of this paper are as follows.

(1) To the best of our knowledge, we are the first one to study metamorphic malware evolution systematically.
(2) Instead of being metamorphic engine specific, our approach is a generalized solution by automatically comparing the possible mutations and memoizing semantically equivalent basic blocks. Our exploration results provide a comprehensive understanding of the metamorphic engine semantics.
(3) We present MetaHunt, a novel approach to comparing the similarities before/after metamorphic mutation. MetaHunt integrates the advanced semantics-based binary diffing technique in metamorphic malware analysis and improves it with better accuracy and performance.

The rest of the paper is organized as follows. Section 2 provides background information and related work. Section 3 and Section 4 present our system design and implementation in detail. We evaluate MetaHunt in Section 5. Discussions and limitations are presented in Section 6. We conclude the paper in Section 7.

2 BACKGROUND AND RELATED WORK

2.1 Metamorphic Malware

Metamorphic malware mutates their code during each generation so that the new generated version reveals different instructions with the previous one, but the semantics is preserved. This differs from polymorphic malware (e.g., via binary packing) which do not rewrite their own code [37]. The constantly changing property makes it difficult for signature-based anti-malware approaches to recognize all the mutations of the same metamorphic malware. The core of metamorphic malware is a metamorphic engine, which performs a set of transformations to mutate the code. The commonly used code morphing methods are register swapping, instruction substitution, instruction reordering, junk code insertion, and control flow obfuscation (e.g., opaque predicates and control flow flattening). We refer the reader to the literature [37, 47] for more detailed information. As shown in Figure 2, MetaPHOR [19] substitutes one instruction with a set of semantics-persevering instructions; Lexotan32 mutates its code by inserting junk code (the instructions are in italics) and reordering instruction. Note that after mutation, the original single basic block in Figure 2(b) has been divided into multiple basic blocks.

Note that among the multiple mutation methods, instruction substitution is the most sophisticated one. Due to the cumbersome x86 ISA, checking whether two instruction sequences are semantically equivalent is challenging. The advanced semantics-based binary diffing has to rely on symbolic execution and theorem proving techniques to match equivalent instructions. Typically, a metamorphic engine performs code substitution by comparing instructions against a fixed table containing alternative sequences, and then randomly chooses one. Figure 2(a) presents a part of MetaPHOR code substitution table. On the other hand, the pure equivalent instruction substitution rules are not unlimited either [33, 50]; that is, the length of code substitution table is fixed. All of these observations form the basis of our approach.

In Section 1, we have introduced the existing metamorphic malware analysis work. However, a systematical study of metamorphic malware evolution is still missing. Understanding how a morphing engine mutates code over time without a priori knowledge is an

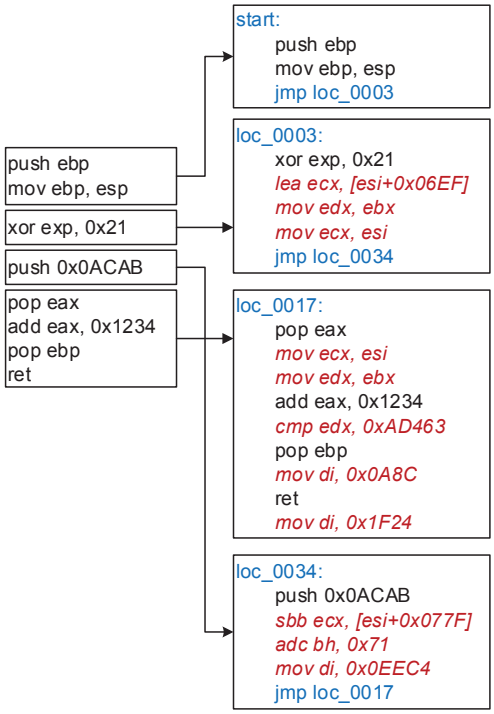
Before	After
jmp eax	push eax ret
xor eax, -1	not eax
and eax, 0	mov eax, 0
add eax, 1	not eax neg eax

movsb	push eax
	mov al, [esi]
	inc esi
	mov [edi], al
	inc edi
	pop eax

stosb	mov edi, [al] inc edi

mov [ebx+8], eax	push ecx
	mov ecx, eax
	mov [ebx+8], ecx
	pop ecx

(a) MetaPHOR [19]



(b) Lexotan32 [38]

Figure 2: Metamorphism transformation examples.

interesting and challenging research problem. In this paper, we propose MetaHunt to explore this problem.

2.2 Semantics-based Binary Diffing

Since most malware spread in binary form, the techniques to detect the difference between two binaries (*binary diffing*) have been widely applied to malware reverse engineering. Conventional binary diffing tools identify syntactical differences such as instruction sequences, byte N-grams, and basic block hashing [36]. However, they can be easily evaded by various obfuscation methods. The core method of the advanced semantics-based binary diffing [21, 29, 30, 35] is to first identify semantically equivalent basic block pairs. It uses symbolic values to represent inputs to a basic block and then simulates the function of each instruction by updating the corresponding symbolic formula. The output of symbolic execution is a set of formulas that represent the behavior of the basic block. After that, we try to find whether there is an equivalent mapping between two basic block output formulas. If yes, those two basic blocks are equivalent in semantics. Figure 3 presents two semantically equivalent basic blocks. Their output symbolic formulas are verified as equivalence by a constraint solver (e.g., STP [20]). Note that due to obfuscation such as register renaming, basic blocks could use different registers or variables to implement the same functionality. As a result, current approaches exhaustively try all possible pairs to find if there exists a bijective mapping between output formulas. However, due to the slow symbolic execution and the high invocation of a constraint solver, semantics-based binary diffing suffers from significant performance slowdown [29].

The most relevant work to MetaHunt is the memoized binary diffing method [32], another trace-oriented binary diffing tool for matching basic block pairs. However, MetaHunt is designed for comparing a large number of obfuscated metamorphic malware variants; the binary diffing method [32] is used for comparing different versions of normal programs. Compared to it, MetaHunt is augmented with better resilience to various code obfuscation methods (e.g. call/return obfuscation and opaque predicate) and a set of optimizations. Therefore, MetaHunt has better accuracy and performance on analyzing metamorphic malware.

3 SYSTEM DESIGN

3.1 Overview

The architecture of MetaHunt is illustrated in Figure 4. It mainly comprises two parts: online trace logging and offline comparison. The online part will produce a sequence of executed basic blocks together with their associated taint tags, and then pass them to the offline part for comparison. MetaHunt's offline stage consists of three components: normalization, basic block comparison with the semantics-based binary diffing technique, and a union-find set structure to record semantically equivalent basic blocks. The normalization component performs several transformations to remove

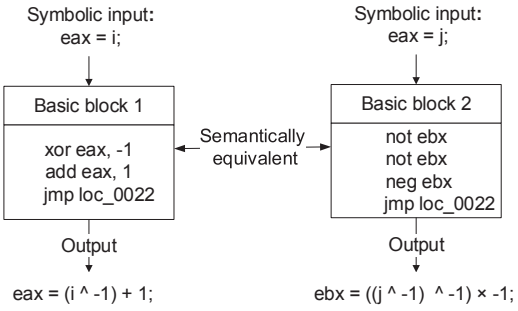


Figure 3: Example: basic block symbolic execution.

obfuscation effect. After that, the normalized basic blocks are compared by a symbolic execution based method. Finally, the equivalent basic blocks are inserted into the same union-find set. The detail of each component are discussed in the following sections.

3.2 Trace Logging

The online trace logger records the basic blocks executed during runtime. In general, not all of the executed instructions are of interest, such as the code from packers or standard libraries. We want to compare the basic blocks that represent the virus behavior. Our online stage supports recording the execution trace that comes from real payload instead of various unpacking routines [45]. When a packed binary starts running, the generic unpacking plug-in will be invoked to monitor whether the original code is recovered; if so, the trace logging plug-in will be activated to record execution trace. Moreover, usually different metamorphic variants still call standard libraries, but the basic blocks in these libraries should not be compared. Our trace logger only records the code from the metamorphic virus ignoring the standard library calls.

In addition to ignoring the unrelated basic blocks during run time, we also limit our comparison to the *input-related code*. The insight is that the basic blocks related to inputs implement the core function of a virus, so these basic blocks should be recorded and compared. To this end, we utilize multi-tag taint forward tracking to record input-related code, which also reduces the number of possible basic block matches. We not only take multiple system calls that are used to receive outside input as different taint seeds but also consider the system calls that are commonly used to fulfill malicious behavior, such as download and execution, replication and remote injection. For example, when a MetaPHOR version executes, it invokes about 20 Windows Native API calls[3] for replicating and displaying its messages. Note that for the file-infecting metamorphic viruses (e.g., MetaPHOR and W32.Evol), multi-tag taint tracking can also distinguish the host file code and virus body code. The input-related basic blocks together with their associated taint tags will be passed to the MetaHunt's offline stage for comparison.

3.3 Basic Block Normalization

After logging the execution trace, MetaHunt first lifts x86 instructions to an intermediate representation (IR), which facilitates the following analysis. The comparison unit of most semantics-based binary diffing work is basic block [21, 29, 30]. However, many obfuscation methods can split a single basic block to multiple basic blocks. As a result, direct comparison between the split basic blocks with the original block lead to false negatives. Moreover, too much extra basic block comparisons increase the performance cost. Therefore, a normalization pass is performed to reverse these obfuscation effects. Currently, we consider three major obfuscation methods: instruction reordering, call/return obfuscation, and opaque predicate obfuscation. The effect of instruction reordering is to split one basic block into multiple new basic blocks, which are connected through direct jumps. `call/return` obfuscation involves non-standard use of the `call` and `ret` instructions [45]. For example, `push ADDR; ret` is equivalent to `jmp ADDR`. Reverting the effect of instruction reordering or call/return obfuscation is straightforward. We merge all adjacent basic blocks that have only one predecessor and one successor into a single basic block.

Our normalization also removes opaque predicate obfuscation. An opaque predicate means its value is known to the obfuscator at obfuscation time, but it is difficult for an attacker to figure it out afterward. For example, predicate ($x^3 - x \equiv 0 \pmod 3$) in Figure 5 is true for all integers x. Opaque predicates have been widely used to introduce redundant branches for the purpose of control flow obfuscation [31]. To handle opaque predicates, we submit a branch condition to a constraint solver to verify whether it is always true or false. If yes, we conclude that the branch condition is an opaque predicate. After that, as shown in Figure 5, the unreachable paths and redundant predicates will be discarded; the basic blocks split by the opaque predicate will be merged.

In addition, we also normalize basic blocks to ignore offsets that may change due to code relocation and some `nop` instructions. Binary code compiled from the same source code often have different address value caused by memory relocation during compilation. What's more, malware authors may intentionally insert some instruction idioms like `nop` and `xchg eax, eax` to mislead the following hash value calculation (see Section 3.4). The purpose of normalization is to ignore these effects and make the hash value more general.

3.4 Basic Block Comparison and Memoization

The basic blocks tainted by the same taint tags are the candidates to be compared. Our basic block comparison is based on semantics-based binary diffing with improvements in several ways. First, we introduce an union-find set structure that records semantically equivalent basic blocks. Managing the union-find structure during successive comparisons allows direct reuse of previously computed results rather than comparing them again. Specifically, after basic block normalization, we first calculate the MD5 value of the byte sequence of each basic block. Then, we dynamically maintain a set of union-find subsets to record semantically equivalent basic blocks, which are represented by their MD5 value. The basic blocks within the same subset are all semantically equivalent to each other. To avoid a highly unbalanced searching tree, we adopt an improved path compression and weighted union algorithm [12]. In addition to the union-find set, we also maintain a `DiffMap` to record two subsets that have been verified that they are not equivalent. If two

[3]The system calls in Windows are named as *Native API*.

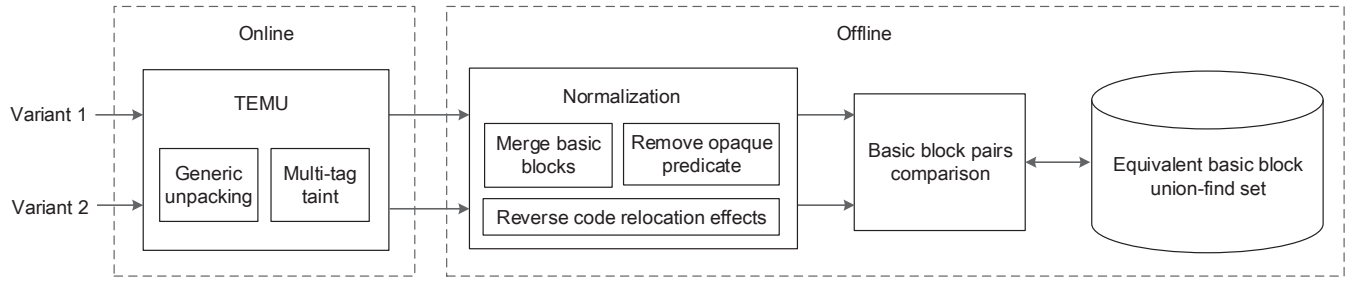


Figure 4: The architecture of MetaHunt.

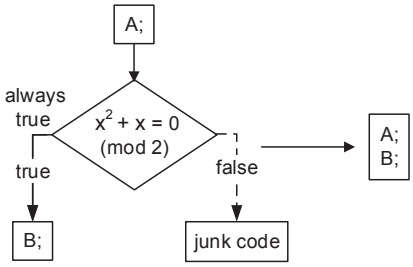


Figure 5: Remove opaque predicate (x is an integer).

basic blocks residing in different subsets are not equivalent, we can safely conclude that the left basic blocks in these two subsets cannot be matched either. Note that the basic blocks within the same union-find set are the mutations mainly caused by instruction substitution transformations, which is confirmed by our evaluation data.

Algorithm 1 presents the method for fast comparing the basic block variants. When comparing two basic blocks, we first normalize them and compare their hash value (Line 4). This step quickly filters out basic blocks with quite similar instructions. If two hash values are not equal, we will identify whether they belong to the same union-find subset (Line 7). Basic blocks within the same subset are semantically equivalent to each other. If they are in the two different subsets, we continue to check DiffMap to find out whether these two subsets have been ensured not equivalent (Line 10). At last, we have to resort to comparing them with symbolic execution and STP, which is accurate but computationally more expensive. After that we update the union-find set and DiffMap accordingly (Line 17~20).

Algorithm 1 A Fast Comparison of Basic Block Variants

```
    v1, v2 : two basic block variants
 1: function FastCompare(v1, v2)
 2:     v1' ← Normalize(v1)
 3:     v2' ← Normalize(v2)
 4:     if MD5(v1') = MD5(v2') then
 5:         return True
 6:     end if
 7:     if Find(v1') = Find(v2') then // within the same subset
 8:         return True
 9:     end if
10:     if v1' and v2' in DiffMap set then
11:                         // semantically different subsets
12:         return False
13:     else
14:         if Sym_Exec(v1') ~ Sym_Exec(v2') then
15:                         // v1', v2' are semantically equivalent
16:             Union(v1', v2')
17:             Update DiffMap
18:             return True
19:         else            // v1', v2' are not semantically equivalent
20:             Add DiffMap(Find(v1'), Find(v2'))
21:             return False
22:         end if
23:     end if
24: end function
```

4 IMPLEMENTATION

We have implemented MetaHunt on top of BitBlaze [46], a binary analysis platform. MetaHunt's trace logging is built on TEMU, a whole-system emulator for dynamic analysis in BitBlaze [46]. The online part involves two plug-ins: *generic unpacking* and *multi-tag taint tracking*. TEMU is also used as a malware execution sandbox in our evaluation. We extend TEMU to perform multi-tag taint analysis on system calls. We intercept the system calls if they are input-related, and then assign taint tags to the return values. Various taint sources are labeled with different taint tags. The online logging part contains 2, 200 lines of code added/modified in TEMU. MetaHunt's offline stage is based on Vine, BitBlaze's static analysis platform, with 2, 900 OCaml lines of code. Our data flow analysis to get rid of junk code is an extension to Vine's chopping module, and the theorem prover is STP [20]. The RISC-like style and static single assignment (SSA) format of Vine's Intermediate Representation fits the requirement of our analysis. It has a feature to represent many functionally equivalent instructions (e.g., `xor eax, eax` and and `eax, 0`) in the same way, which is extended for the normalization component. The saving and loading of union-find set and query hash map are developed using the OCaml Marshal API, which encodes arbitrary data structures as sequences of bytes and then stores them in a disk file. We also write 500 lines of Perl scripts to glue all components together to automate the comparison process.

5 EVALUATION

We evaluate MetaHunt with several objectives in mind. First, we want to evaluate our iterative comparison of metamorphic variants will converge in a reasonable time, that is, MetaHunt is capable of exploring the morphing code evolution. At the same time, we make sure MetaHunt's exploration results are comprehensive and accurate. We provide a case study of the metamorphic engine in MetaPHOR and W32.Evol to show more details about the engine's mechanism and how MetaHunt explores the variants generated by the engine. We also test the optimization methods for speeding up the malware comparison. At last, we report some interesting findings during our evaluation.

5.1 Experiment Setup

Our testbed consists of Intel Core i7-3770 processor (Quad Core with 3.40GHz) and 8GB memory, running Ubuntu 12.04. The guest OS running in TEMU is Windows XP SP3. The dataset used for our experiment consists of a total 1, 400 metamorphic variants. They are generated by nine mutation engines collected from different sources. Table 1 shows our dataset statistics and the morphing engine information. The second column indicates whether the morphing engine is attached by the malicious body or decoupled. The third column presents the number of metamorphic variants we generate. Column 4 ∼ 13 represent the major code morphing methods adopted by these engines. The morphing methods include register renaming, dead code insertion, instruction reorder, equivalent code substitution, opaque constant, call/return obfuscation, indirect jump, opaque predicate, control flow graph flattening, and function inlining. In addition, some metamorphic viruses also integrate polymorphic encryption. For example, when the binaries infected by Lexotan32 or MetaPHOR execute, the main virus body would be first decrypted [38, 43]. We mark these two cases in the 14th column (Decryption) of Table 1.

As shown in Table 1, the nine mutation engines in our experiment are categorized into three groups. The first group contains three well-known self-propagating viruses, Lexotan32, MetaPHOR, and W32.Evol. They all embed the metamorphic engine within the virus body. In our study, we select 100 copies of Cygwin[4] utility `bzip2` as the "goat" binaries, and the metamorphic virus in the first category are used to infect them. Since these three viruses do not mutate their host code, choosing the same copies of goat files can help us identify the morphing code, which is always being modified while the host code does not change. During our evaluation, the running goat executables will infect themselves iteratively, and each infection will yield a new generation variant. The sizes of the malware samples in this group range from 28 KB to 410 KB. Virus construction kits are designed to simplify the development of virus code, and some kits are also used as decoupled metamorphic engines to mutate non-propagating malware [47]. The second group in Table 1 consists of four mutator cases, which are collected from VX Heaven[5]. For each tool, we create 200 metamorphic virus variants. Since the output of these virus generators is assembly code, we use TASM 5.0 Assembler to compile the source code into binary. The sizes of the viruses in this group vary from 1 KB to 300 KB. The third group are two open source metamorphic generators based on LLVM: MalDiv [41] and Obfuscator-LLVM [24]. They perform code mutation by manipulating the LLVM IR code. We select the source code of Linux utility `gzip` as the base file. We generate the metamorphic variants of `gzip` by applying the MalDiv and Obfuscator-LLVM iteratively on the mutated code. The sizes of the `gzip` mutations in this group range from 61 KB to 728 KB.

[4]https://www.cygwin.com
[5]http://vxheaven.org

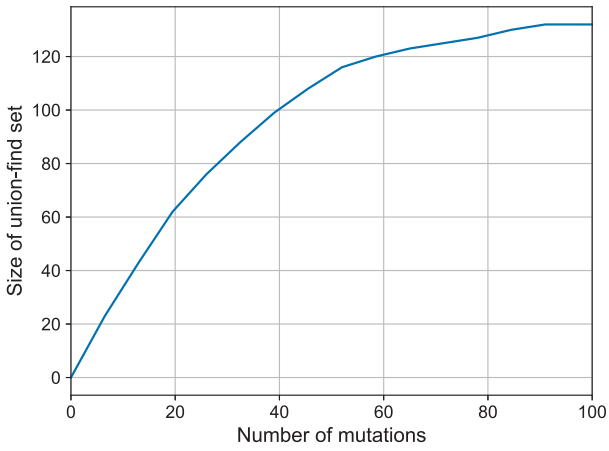


Figure 6: Size of union-find set of the variants generated by MetaPHOR.

5.2 Converging Time and Union-Find Set Size

We run MetaHunt to compare the metamorphic variants in Table 1 and the result is reported in column 15∼17. Column 15 shows the converging time of our iteratively comparing metamorphic mutations. Column 16 and 17 present statistics of the union-find set, including the number of union-find subsets and the maximum number of basic blocks in one subset. The result in Figure 6 shows that MetaHunt reaches a converging point in 7 hours for all the metamorphic engines. For these variants from simpler engines such as Lextan32, W32.Evol, G2, and VCL32, MetaHunt takes less than 2 hours to reach the converging point. After reaching the converging point, the size and number of union-find set in MetaHunt stop growing, which means MetaHunt has studied the evolution of the variants from the metamorphic engine. We also record the number of union-find subsets and the maximum number of basic blocks in one set. We looked into the basic blocks inside one subset, and manually verified that they are all semantically equivalent variants of the same basic block. Our evaluation result shows the mutation capability of metamorphic malware is not unlimited, and the evaluation of variants will eventually reach to a converge point. Consider the number of variants is unable to increase continuously, this may provide a start point for the malware defender and stimulate designing mutation insensitive anti-malware solutions.

5.3 Case Study

5.3.1 MetaPHOR. We analyze the source code of MetaPHOR virus(version 1.1), which was first published on a virus and worm

Table 1: Metamorphic engine statistics and various code mutation methods adopted.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Engine	Type	# Mutations	Reg. renaming	Dead code	Instr. reorder	Instr. substitut.	Opaque constant	call/return obfus.	Indirect jump	Opaque pred.	CFG flattening	Funct. inlining	Decryption	Conv. time (hrs)	# UF subsets	Max. subset size
Lexotan32	attached	100	✓	✓	✓	✓				✓			✓	1.5	90	8
MetaPHOR	attached	100	✓	✓	✓	✓				✓			✓	2.2	132	12
W32.Evol	attached	100	✓	✓	✓	✓				✓				1.0	52	6
NGVCK	decoupled	200	✓	✓	✓	✓	✓	✓	✓	✓				4.7	346	16
G2	decoupled	200	✓	✓	✓	✓								1.4	115	8
VCL32	decoupled	200	✓	✓	✓	✓								1.8	130	10
MPCGEN	decoupled	200	✓	✓	✓	✓								2.2	96	8
MalDiv	decoupled	150	✓	✓	✓	✓	✓	✓	✓	✓				6.8	522	34
Obfuscator-LLVM	decoupled	150	✓	✓	✓	✓	✓			✓	✓	✓		4.6	304	18

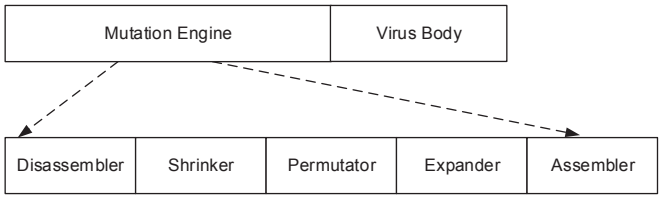


Figure 7: Structure of the MetaPHOR virus

group 29A[6]. It has about 14,000 lines of assembly code. As shown in Figure 7, the structure of MetaPHOR consists of two parts, metamophic enigne and the virus body.

Unlike other metamorphic viruses, MetaPHOR has a complete metamorphic engine, which provides full support for absolute metamorphism. The metamorphic engine includes five components: disassembler, shrinker, permutator, expander, and assembler. The disassembler first decodes the virus body and transforms its instructions into pseudo-assembly language, which benefits the following mutation steps finished by other components of the engine. Shrinker is responsible of compressing the disassembled code preprocessed by the disassembler, in order to avoid explosive growth of code size in very few generations. The permutator further mutates the code by redefining the "code frame" size and shuffling the code frame sequence. As the opposite component of the shrinker, expander does what the shrinker undoes, which recodes a single instruction to many instructions that perform the same function. At last, the assembler will reassemble the pseudo-assembly code back to the machine code.

Based on our analysis, the function layout of the source code is as follows. The disassembler part is between line 1520 and line 3041. The shrinker part resides in lines 3049 and 5765. The permutator part starts from line 5929 and ends at line 6413. The expander part is on lines between 6453 and 9279. The lines between 9306 and 10485 is for the assembler. Besides, there are some other parts, infector (the virus body) and polymorphic engine. The infector is on lines 10541 to 12578. Polymorphic engine code is between lines 12621 and 13887. The rest of the code are some other minor features.

In each propagation, MetaPHOR will first reserve a 0x340000 bytes space for the next-generation variant, and this space is fixed. This partially ensures that the generations of MetaPHOR will not go out of control on the code size. Besides, the shrinker and expander use transformation tables to compress or expand the virus body, and the transformation tables are fixed. Figure 2(a) shows the transformation examples used by the expander, while transformation examples used in shrinker is demonstrated in Figure 9. When producing a next-generation variant, both shrinker and expander are able to process the virus instructions for multiple rounds. The virus body will change in each round literately, and after each round all the transformation will be accumulated in the next-generation variant.

Between shrinker and expander is permutator. The algorithms used in permutator includes redefining "code frames" and shuffling. The first step is redefining "code frames". Given an initial and a final offset, new "code frame" sizes are selected randomly between F0h and 1E0h until the last "code frame" reach to the end of code. All the new "code frame" entries will be stored in a table. Then, the "code frame" sequence in the table will be shuffled. After the shuffling process, the permutator will start copying the instructions according to the shuffled sequence of "code frames". At last, the JMP instructions will be inserted at the end of new "code frame", and at the same time the virus behavior will be unchanged. Figure 8 shows a permutation example. These basic block transformations are listed to show the typical mutation methods used in the metamorphic malware samples. Since the permuted basic blocks are connected by unconditional jumps, they are normalized to one basic block in MetaHunt's analysis. In fact, the permutation transformation is removed by the normalization component and it does not affect the binary diffing in MetaHunt. Therefore, MetaHunt is able to reverse the change made by the permutator.

Based on our analysis, we found MetaPHOR is capable of mutating itself. However, the mutation is not infinite, which properly

[6]http://virus.wikidot.com/metaphor

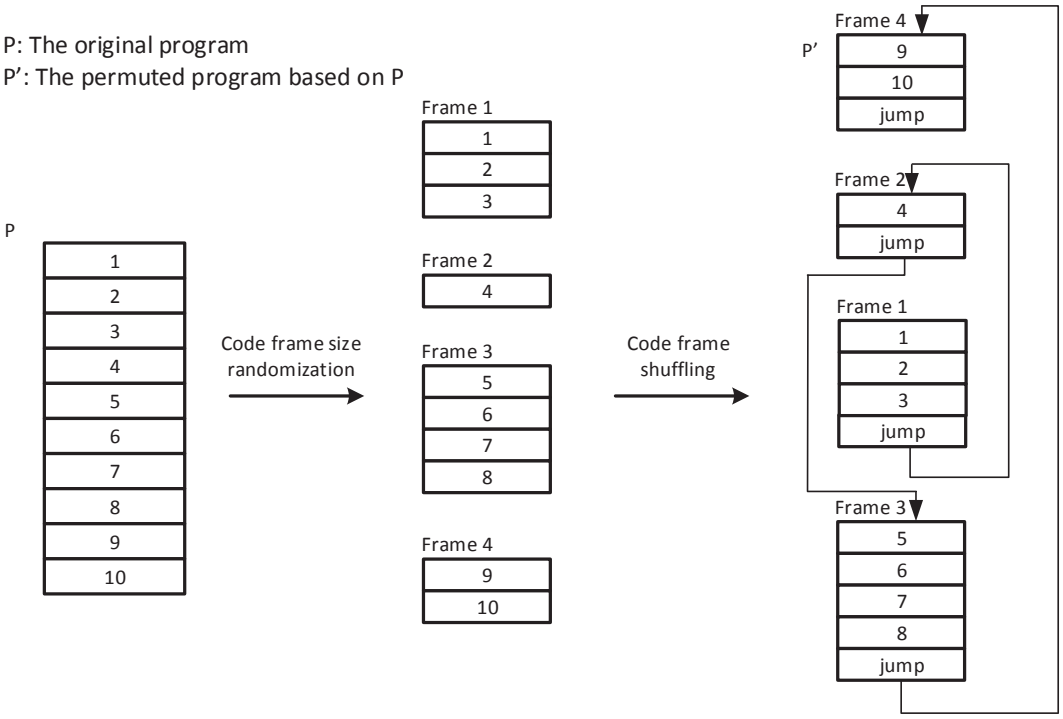


Figure 8: An example of one-time permutation

Before	After
mov eax, 1 add eax, ecx	lea eax, [ecx+1]
push 3 pop eax	mov eax, 3
mov eax, ebx add eax, 8	lea eax, [ebx+8]
mov [eax], 3 push [eax]	push 3

mov [eax], ebx
add [eax], ecx
mov ebx, [eax]
	add ebx, ecx

mov [eax], 2
add [eax], ecx
mov ebx, [eax]
	add ecx, 2

or eax, 0	nop

Figure 9: Code compressing examples in MetaPHOR

explains our experiment results. Since the metamorphic engine uses fixed transformation tables (in shrinker and expander) and the reserved space for virus body is fixed, we can conclude that although MetaPHOR employs full metamorphism engine, it only has a finite length of evolution, which can be studied by MetaHunt.

5.3.2 W32.Evol. The W32.Evol virus is first discovered in July 2000[7], which is the first virus to utilize a 'true' 32-bit metamorphic engine instead of the polymorphic engine which is susceptible to AV scanners that can trace virus decryption in memory. A metamorphic engine is used to transform the executable code: it implements some sort of an internal disassembler to parse input code, and then transforms the program code and produces new different code while retaining its functionality.

The instruction transformation supported by the engine can be divided into two parts: Inter-engine transformations are inlined inside the engine as a part of the engine's core. External Transformations take place outside the main engine function, yet they act as if they are inside the engine itself and jump back to the engine when they are finished. The engine's decision on whether or not to transform a given instruction is based upon a random factor. The engine asks for a random number between 0 and 7, and the transformation will be applied only if it is 0. Hence there is a probability of 12.5% that an instruction would be transformed. Furthermore, the engine will only disassemble the instructions that the author had included.

As shown in Figure 10, the disassembly of the virus' code before transformation in the left column and the corresponding transformed code in the right column. We can see that for the first row, the transformation is semantics-preserving unconditionally. However, for the last two rows, we can see the value of `eax` is given the value `0x04` and `0x09` respectively. Therefore, the last two transformations

[7] https://www.symantec.com/security-center/writeup/2000-122010-0045-99

Before	After

push eax
mov [edi], 0x03
	push eax
	push ecx
	mov ecx, 0x03
	mov [edi], ecx
	pop ecx

push 0x03
mov eax, 0x08
	mov eax, 0x03
	push eax
	mov eax, 0x08

mov eax, 0x03
push eax
	mov eax, 0x03
	push eax
	mov eax, 0x08

Figure 10: An example of conditional transformation in W32.Evol

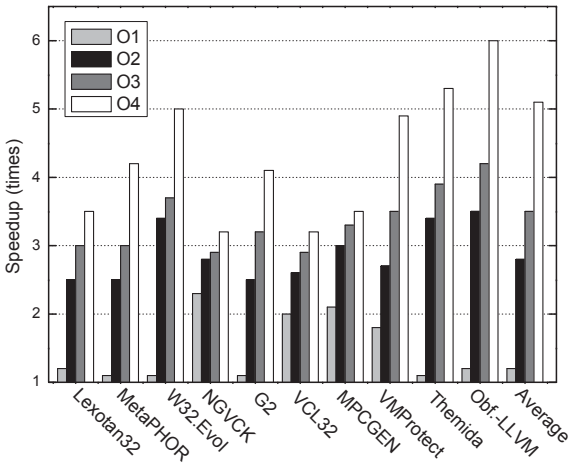


Figure 11: The impact of basic blocks fast matching when applied cumulatively: O1 (preprocessing), O2 (O1 + union-find set and DiffMap), O3 (O2 + concretizing symbolic formulas), O4 (O3 + QueryMap).

are semantics-preserving only if the register eax is not live when they are applied.

Similar to MetaPHOR, the W32.Evol engine adopts a fixed transformation table. Moreover, because the W32.Evol engine only disassembles instructions that the virus author included, and the conditional transformation can only be applied at the point certain registers are not live, these restrictions further limits the possible mutations of the W32.Evol virus, which is coherent to our experiments.

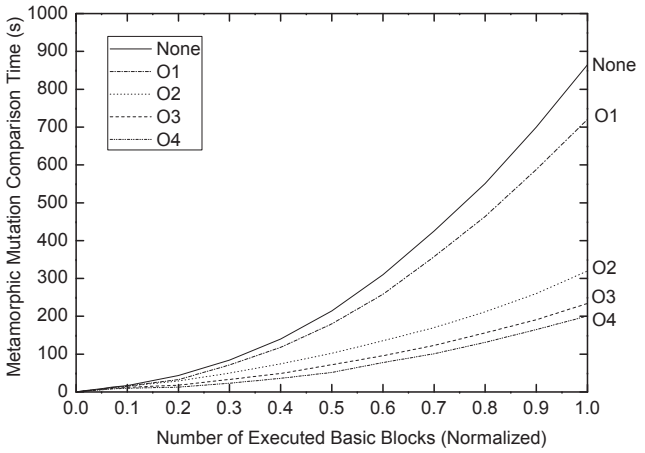


Figure 12: The effect of our optimizations over time on NGVCK family

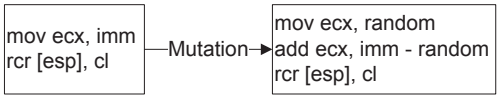


Figure 13: Example: buggy metamorphic engine implementation (add instruction may modify the value of carry flag).

5.4 Optimization

The binary comparison component in MetaHunt is optimized for quickly checking the equivalence of two basic blocks. Various methods in MetaHunt contribute to improve the performance of comparison. First, the preprocessing normalizes the trace and remove the obfuscations. Second, the union-find set and DiffMap keep the checked basic block in memory so as to accelerate the future comparison. Third, concretizing the formulas in symbolic execution replaces some unnecessary symbols with concrete values. It reduces the complexity of symbolic formulas so that the comparison is faster. Last, the querymap in MetaHunt calculate hash value of the symbolic formulas and use them to quickly match the formulas. In order to show the speedup effect of each of these methods, we incrementally add one method in MetaHunt and use it to compare the variants in all the metamorphic engines. The result of the experiment is shown in Figure 11. We can see that every method has a significant speed-up and it achieves over 5X speedup with all optimizations turned on.

We conduct another experiment on the NGVCK metamorphic virus family and the result is shown in Figure 12. We observe that the optimizations in MetaHunt are able to speed up the comparison, especially when there are large number of basic block candidates. Therefore, MetaHunt's optimization can improve its performance on comparing basic block variants from metamorphic engines.

5.5 Finding Metamorphic Engine Bugs

Most metamorphic malware are running on the Intel x86 platform because of its popularity. However, x86 Instruction Set Architecture

Table 2: Conditionally equivalent instructions (reg, imm and random stand for register, immediate value and random number, respectively).

Instruction	Substitution	Condition
inc reg	add reg, 1	carry flag is not set
dec reg	sub reg, 1	carry flag is not set
pop reg	mov reg, [esp] add esp, 4	no EFLAGS bit is set
push reg	sub esp, 4 mov [esp], reg	no EFLAGS bit is set
add reg, imm	sub reg, -imm	overflow and carry flags are not set
mov reg, imm	mov reg, random add reg, imm - random	no EFLAGS bit is set

is complicated as well, which make the design of metamorphic transformation rules very difficult. Especially, certain instructions have implicit side effects. They reveal different semantics when the value of EFLAGS register varies. If a metamorphic engine neglects such subtleties of x86 instructions, it is very likely that semantics-breaking mutations will happen. Table 2 lists that some instructions and their substitutions are only *conditionally equivalent* when certain EFLAGS register bits are dead. For example, the Intel manual indicates that "inc/dec" does not affect the carry flag while "add/sub" does; the instruction "pop reg" (the third row in Table 2) does not modify any EFLAGS bits while "add" may set as many as six bits. Unfortunately, the examples shown in Table 2 are misused by many of our testing metamorphic engines. Figure 13 shows a possible semantics-breaking mutation we find in NGVCK. The instruction "rcr" rotates right using the carry flag as the "extra" bit. Therefore, the modification to the carry flag before the "rcr" instruction may lead to an incorrect rotation result. However, the "add" instruction in the new version may modify the value of carry flag. Since MetaHunt also trace the symbolic execution for each EFLAGS register bit, we can find metamorphic engine bugs in terms of conditionally equivalent transformations. In our evaluation, we find 62 semantics-breaking bugs in total. These metamorphic engine bugs lead to fatal runtime errors in many cases.

6 DISCUSSIONS AND LIMITATIONS

The power of MetaHunt is limited by the non-perfect path coverage. This is mainly due to the limitation of dynamic malware analysis. We can leverage automatic input generation techniques [22] to explore more paths. Since MetaHunt depends on multi-tag taint analysis to reduce the number of basic block comparisons, MetaHunt exhibits similar limitations of taint analysis in general, e.g., implicit information flow evasions [7]. One possible solution is to leverage statistical binary similarity comparison [15, 16] to reduce the number of constraint solving on multiple paths. Another threat to dynamic malware analysis is environment-sensitive malware. Since we analyze metamorphic malware in TEMU, a malware sample can detect itself running in an emulator instead of the physical machine and then quit immediately. To evade such sandbox environment check, a possible countermeasure is to analyze malware in a transparent analysis platform via hardware virtualization (e.g., Ether [18]). Currently, MetaHunt's detection on opaque predicates focuses on invariant opaque predicates, whose value remain the same for all possible inputs. The most recent work can detect more advanced cases such as contextual and dynamic opaque predicates [31]. Although we do not see such complicated opaque predicates in our evaluation, we will extend our work to handle the advanced opaque predicates proactively.

Another argument against studying the evolution of metamorphic malware is the relatively high cost. In fact, compared to the number and diversity of the malware samples in circulation, the metamorphic engine evolves rather slower because of the great development complexity. A successful metamorphic engine tends to be reused and shared by malware authors. For example, NGVCK [55] is widely applied to generate metamorphic virus and Obfuscator-LLVM, is also used to mutate both desktop and Android applications [24, 54]. Therefore, our one-time efforts to approximate the semantics of nontrivial metamorphic engines are worthwhile. Furthermore, considering that manually tracing metamorphic mutations usually takes several days to weeks of hard work, the degree of MetaHunt's overhead is acceptable.

7 CONCLUSION

The metamorphic malware relies on its morphing engine to mutate the malicious code from generation to generation so that each variant is different in syntax. Metamorphic malware have been demonstrated to evade the conventional signature-based malware detection successfully. The mutation engine itself is also constantly evolving. In this paper, we attempt to tame the metamorphic mutation by systematically chasing the morphing code evolution. We apply trace-based semantic binary diffing to compare possible mutation variants iteratively and memoizes equivalent basic blocks. Without pre-knowledge about a particular metamorphic engine, our exploration result can approximate its mutation mechanism. We have implemented our approach called MetaHunt and performed empirical evaluations on a large set of metamorphic malware. Our generalized approach can be seen as a first step towards designing mutation insensitive anti-malware solutions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) grants CNS-1652790, and the Office of Naval Research (ONR) grants N00014-16-1-2265, N00014-16-1-2912, and N00014-17-1-2894. Jiang Ming was also supported by the University of Texas System STARs Program.

REFERENCES

[1] Shahid Alam, Issa Traore, and Ibrahim Sogukpinar. 2014. Current Trends and the Future of Metamorphic Malware Detection. In *Proceedings of the 7th International Conference on Security of Information and Networks (SIN'14).*

[2] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13).*

[3] Philippe Beaucamps. 2007. Advanced Metamorphic Techniques in Computer Viruses. In *Proceedings of the 2007 International Conference on Computer, Electrical, and Systems Science, and Engineering (CESSE'07).*

[4] D. Bruschi, L. Martignoni, and M. Monga. 2006. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'06).*

[5] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2007. Code Normalization for Self-Mutating Malware. *IEEE Security and Privacy* 5, 2 (2007).

[6] Lorenzo Cavallaro. 2014. Malicious Software and its Underground Economy. https://www.coursera.org/course/malsoftware.

[7] L. Cavallaro, P. Saxena, and R. Sekar. 2008. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'08)*.

[8] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. 2010. SplitScreen: Enabling Efficient, Distributed Malware Detection. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*.

[9] Mohamed R. Chouchane and Arun Lakhotia. 2006. Using Engine Signature to Detect Metamorphic Malware. In *Proceedings of the 4th ACM Workshop on Recurring Malcode (WORM'06)*.

[10] Mohamed R. Chouchane, Andrew Walenstein, and Arun Lakhotia. 2007. Statistical Signatures for Fast Filtering of Instruction-substituting Metamorphic Malware. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM'07)*.

[11] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. 2005. Semantics-aware malware detection. In *Proc. of the IEEE Symposium on Security and Privacy*.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms (Second ed.)*. MIT Press, Chapter 21: Data structures for Disjoint Sets, 498–524.

[13] Mila Dalla Preda, Roberto Giacobazzi, and Saumya Debray. 2015. Unveiling metamorphism by abstract interpretation of code properties. *Theoretical Computer Science* 577 (2015), 74–97.

[14] Mila Dalla Preda, Roberto Giacobazzi, Saumya Debray, Kevin Coogan, and Gregg M Townsend. 2010. Modelling metamorphism by abstract interpretation. In *International Static Analysis Symposium*. 218–235.

[15] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[16] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of Binaries Through Re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[17] Priti Desai and Mark Stamp. 2010. A highly metamorphic virus generator. *International Journal of Multimedia Intelligence and Security* 1, 4 (2010).

[18] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'08)*.

[19] The Mental Driller. last reviewed, 04/14/2015. Metamorphism in practice or How I made MetaPHOR and what I've learnt. http://vxheaven.org/lib/vmd01.html.

[20] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 2007 International Conference in Computer Aided Verification (CAV'07)*.

[21] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically finding semantic differences in binary programs. In *Poceedings of the 10th International Conference on Information and Communications Security (ICICS'08)*.

[22] P. Godefroid, M. Y. Levin, and D. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[23] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference. In *Proceedings of the 22nd USENIX Security Symposium*.

[24] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO'15)*.

[25] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. 2013. Fast Location of Similar Code Fragments Using Semantic 'Juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13)*.

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*.

[27] Felix Leder, Bastian Steinbock, and Peter Martini. 2009. Classification and detection of metamorphic malware using value set analysis. In *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE'09)*.

[28] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. 2010. Detecting Metamorphic Malwares using Code Graphs. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*.

[29] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *Proc. of the 22nd ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering (FSE'14)*.

[30] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary Hunting with Inter-Procedural Control Flow. In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)*.

[31] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicates Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*.

[32] Jiang Ming, Dongpeng Xu, and Dinghao Wu. 2015. Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference. In *Proc. of the 30th IFIP Int'l Information Security and Privacy Conference (IFIP SEC'15)*.

[33] Vishwath Mohan and Kevin W Hamlen. 2012. Frankenstein: Stitching Malware from Benign Binaries. *WOOT* 12 (2012), 77–84.

[34] Vinod P. Nair, Harshit Jain, Yashwant K. Golecha, Manoj Singh Gaur, and Vijay Laxmi. 2010. MEDUSA: MEtamorphic Malware Dynamic Analysis Using Signature from API. In *Proceedings of the 3rd International Conference on Security of Information and Networks (SIN'10)*.

[35] Beng Heng Ng and Atul Prakash. 2013. Exposé: Discovering Potential Binary Code Re-use. In *Proceedings of the 37th IEEE Annual Computer Software and Applications Conference (COMPSAC'13)*.

[36] Jeong Wook Oh. 2009. Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries. In *Proceedings of the 2009 Black Hat USA*.

[37] Philip OKane, Sakir Sezer, and Kieran McLaughlin. 2011. Obfuscation: The Hidden Malware. *IEEE Security and Privacy* 9, 5 (2011).

[38] Orr. last reviewed, 04/14/2015. The Molecular Virology of Lexotan32: Metamorphism Illustrated. http://www.openrce.org/articles/full_view/29.

[39] Rodney Owens and Weichao Wang. 2011. Non-normalizable Functions: a New Method to Generate Metamorphic Malware. In *Proceedings of the 2011 IEEE Military Communications Conference (MILCOM'11)*.

[40] Panda Security. 2017. PandaLabs Annual Report 2017. https://www.pandasecurity.com/mediacenter/src/uploads/2017/11/PandaLabs_Annual_Report_2017.pdf.

[41] Mathias Payer. 2014. Embracing the new threat: towards automatically, self-diversifying malware. Symposium on Security for Asia Network (SyScan'14).

[42] Mathias Payer, Stephen Crane, Per Larsen, Stefan Brunthaler, Richard Wartell, and Michael Franz. 2014. Similarity-based matching meets Malware Diversity. *arXiv Technical Report* (2014).

[43] Frédéric Perriot, Peter Ferrie, and Péter Ször. 2003. Striking Similarities: Win32/Simile and Metamorphic Virus Code. Symantec Security Response.

[44] Mila Dalla Preda. 2012. The Grand Challenge in Metamorphic Analysis. In *Proceedings of the 6th International Conference on Information Systems, Technology and Management (ICISTM'12)*.

[45] Kevin A. Roundy and Barton P. Miller. 2013. Binary-code Obfuscations in Prevalent Packer Tools. *Comput. Surveys* 46, 1 (2013).

[46] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*.

[47] Peter Szor. 2005. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.

[48] Péter Ször and Peter Ferrie. 2001. Hunting For Metamorphic. Symantec White Paper.

[49] Teja Tamboli, Thomas H. Austin, and Mark Stamp. 2014. Metamorphic code generation from LLVM bytecode. *Computer Virology and Hacking Techniques* 10, 3 (2014), 177–187.

[50] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. 2008. Constructing malware normalizers using term rewriting. *Computer Virology* 4, 4 (2008), 307–322.

[51] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*. USENIX Association.

[52] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. Uroboros: Instrumenting Stripped Binaries with Static Reassembling. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*. USENIX Association.

[53] Matt Webster and Grant Malcolm. 2009. Detection of metamorphic and virtualization-based malware using algebraic specification. *Computer Virology* 5, 3 (2009), 221–245.

[54] Ryan Welton. 2015. Obfuscating Android Applications using O-LLVM and the NDK. http://fuzion24.github.io/.

[55] Wing Wong and Mark Stamp. 2006. Hunting for metamorphic engines. *Computer Virology* 2, 3 (2006), 211–229.

[56] Qinghua Zhang and Douglas S. Reeves. 2007. MetaAware: Identifying Metamorphic Malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*.