Noname manuscript No.
(will be inserted by the editor)

MalwareHunt: Semantics-Based Malware Diffing Speedup by Normalized Basic Block Memoization

Jiang Ming · Dongpeng Xu · Dinghao Wu

the date of receipt and acceptance should be inserted later

Abstract The key challenge of software reverse engineering is that the source code of the program under investigation is typically not available. Identifying differences between two executable binaries (binary diffing) can reveal valuable information in the absence of source code, such as vulnerability patches, software plagiarism evidence, and malware variant relations. Recently, a new binary diffing method based on symbolic execution and constraint solving has been proposed to look for the code pairs with the same semantics, even though they are ostensibly different in syntactics. Such semantics-based method captures intrinsic differences/similarities of binary code, making it a compelling choice to analyze highly-obfuscated malicious programs. However, due to the nature of symbolic execution, semantics-based binary diffing suffers from significant performance slowdown, hindering it from analyzing large numbers of malware samples. In this paper, we attempt to mitigate the high overhead of semantics-based binary diffing with application to malware lineage inference. We first study the key obstacles that contribute to the performance bottleneck. Then we propose *normalized basic block memoization* to speed up semantics-based binary diffing. 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. Moreover, we utilize a set of enhanced optimization methods to further cut down the invocation numbers of constraint solver. We have implemented our technique, called *MalwareHunt*, on top of a trace-oriented binary diffing tool and evaluated it on 15 polymorphic and metamorphic malware families. We perform intra-family comparisons for the purpose of malware lineage inference. Our experimental results show that MalwareHuntcan accelerate symbolic execution from 2.8X to 5.3X (with an average 4.1X), and reduce constraint solver invocation by a factor of 3.0X to 6.0X (with an average 4.5X).

Keywords: binary diffing, semantics, symbolic execution, malware lineage inference, normalized basic block memoization

1 Introduction

In many tasks of software security, the source code of the program under examination is typically absent. Instead, the executable binary itself is the only available resource to analyze. Therefore, determining the real differences between two executable binaries has a wide variety of applications, such as latent vulnerabilities exploration [26], automatic "1-day" exploit generation [2] and software plagiarism detection [22]. Conventional approaches can quickly locate syntactical differences by measuring instruction sequences [34], byte N-grams [18] or fingerprint hashing [28]. However, such syntax-based

A preliminary version of this paper [25] appeared in the *Proceedings of the 30th IFIP TC-11 SEC International Information Security and Privacy Conference (IFIP SEC'15), Hamburg, Germany, May 26-28, 2015.*

Jiang Ming
The Pennsylvania State University, University Park, PA 16802, U.S.A.
E-mail: jum310@ist.psu.edu

Dongpeng Xu
The Pennsylvania State University, University Park, PA 16802, U.S.A.
E-mail: dux103@ist.psu.edu

Dinghao Wu
The Pennsylvania State University, University Park, PA 16802, U.S.A.
E-mail: dwu@ist.psu.edu

comparison can be easily evaded by various obfuscation techniques, such as instruction substitution [15], binary packing [33], and self-modifying code [3]. The latest binary diffing approaches [14, 23, 22, 27] simulate semantics of a snippet of binary code (e.g., basic block) by symbolic execution and represent the input-output relations as a set of symbolic formulas. Because of various code obfuscation methods, the generated formulas could be quite complicated to analyze. However, by submitting the two formulas to a state-of-the-art constraint solver, we can verify whether they are equivalent. Such obfuscation-resilient comparison captures the intrinsic semantics differences/similarities with low false-positive and false-negative rates.

As the underground industry of malware prospers, malware authors frequently update their malicious code to circumvent security countermeasures. According to the latest annual report of Panda Security labs [30], in 2013 alone, there are about 30 million malware samples in circulation and only 20% of them are newly created. Obviously, most of such malware samples are simple update (e.g., apply a new packer) to their previous versions. Therefore, hunting malware similarities is of great necessity. The nature of being resilient to instruction obfuscation makes semantics-based binary diffing an appealing choice to analyze highly obfuscated malware as well. Unfortunately, the significant overhead imposed by the state-of-the-art approach has restricted its application in large scale analysis, such as malware lineage inference [16], which typically requires pairwise comparison to identify relationships among malware variants. Therefore, an efficient semantics-based malware diffing approach is of great necessity.

In this paper, we first diagnose the two key obstacles leading to the performance bottleneck, namely high invocations of constraint solver and slow symbolic execution. To address these issues, we propose *normalized basic block memoization* to accelerate equivalent basic block matching by reusing previously compared results. When performing malware lineage inference, we observe that malware variants are likely to share common code [20]. A new version may only adopt a different packer or incremental updates. As a result, we exploit code similarity by applying union-find set [8], an efficient tree-based data structure, to record semantically equivalent basic blocks which have already been identified. When comparing two basic blocks, we first perform code normalization to reverse some obfuscation effects that could split a single basic block into multiple ones, such as instruction reordering and opaque predicate. Then the matched basic blocks are stored in a union-find set. Maintaining the union-find structure during successive comparisons allows direct reuse of previous results, without the need for re-comparing them. In addition, to further cut down the high invocation numbers of constraint solver, we also utilize concretizing symbolic formulas and caching equivalence queries.

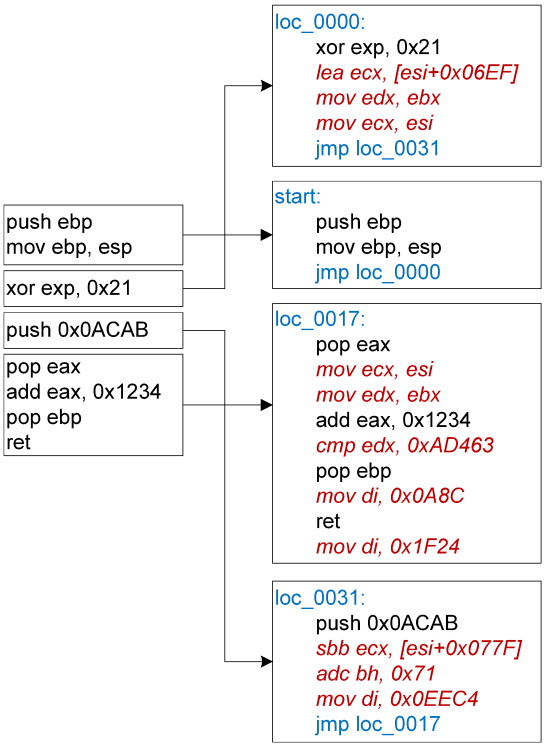
We have implemented our approach, named *MalwareHunt*, on top of iBinHunt [23], a trace-oriented binary diffing tool. We evaluate MalwareHunt in the task of malware lineage inference on 15 malware families, including both polymorphic and metamorphic malware. Our experimental results show that our methods can speed up malware lineage inference, symbolic execution and constraint solver by a factor of 4.4X, 4.1X and 4.5X, respectively. Our proposed solution focuses on accelerating basic blocks matching and can be seamlessly woven into other binary diffing approaches based on equivalent basic blocks. Furthermore, the semantically equivalent basic blocks collected by MalwareHunt can facilitate designing a mutation insensitive anti-malware solution. In summary, the contributions of this paper are as follows.

1. We look into the high overhead problem of semantics-based binary diffing and identify cruxes leading to the performance bottleneck.
2. We propose *normalized basic block memoization* to enable more efficient binary comparison, including maintaining a union-find set structure, concretizing symbolic formulas and caching equivalence queries.
3. We extends the advanced semantics-based binary diffing techniques to analyze obfuscated malware instances and ameliorate the performance bottleneck

The rest of the paper is organized as follows. Section 2 provides the background information. Section 3 studies the performance bottleneck of semantics-based binary diffing. Section 4 describes our optimization methods in detail. MalwareHunt's implementation and evaluation details are presented in Section 5. Related work are introduced in Section 6. At last, we conclude the paper in Section 7.

2 Background

In this section, we introduce the background information of semantics-based binary diffing. Previous binary diffing tools can quickly identify syntactical differences such as instruction sequences [34], byte N-grams [18] and fingerprint hashing [28]. However, they can be easily defeated by various obfuscation methods. For example, Fig. 1 shows two counterexamples of metamorphism transformations. Metamorphic malware mutates its code during infection so that each variant reveals different instructions in syntax. As shown in Fig. 1, Lexotan32 [29] mutates its code by inserting junk code



(a) Lexotan32 [29]

Before	After

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

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

(b) MetaPHOR [10]

Fig. 1 Example: metamorphism transformation.

(the instructions are in italics) and reordering instruction (Fig. 1(a)); MetaPHOR [10] substitutes one instruction with a set of semantics-persevering instructions (Fig. 1(b)). Note that after mutation, the original single basic block in Fig. 1(a) has been divided into multiple basic blocks. The core method of semantics-based binary diffing [14,23,22,27] is to identify semantically equivalent basic block pairs. A Basic block is a straight-line code with only one entry point and only one exit point, which makes the basic block an ideal fit for symbolic execution (e.g., without conjunction of path conditions). Fig. 2 presents a motivating example to illustrate how the semantics of a basic block is simulated by symbolic execution. The two basic blocks in Fig. 2 are semantically equivalent, even though they have different x86 instructions (labeled as bold). In practice, symbolic execution is performed on an RISC-like intermediate language (IL), which represents complicated x86 instructions as simple single static assignment (SSA) style statements. In Fig. 2, the registers have been represented as SSA style (e.g., ecx_0, edx_1).

We take the inputs to the basic block as symbols and simulate the effect of each instruction by updated the corresponding symbolic formula. The output of symbolic execution is a set of formulas that represent input-output relations of the basic block. Now determining whether two basic blocks are equivalent in semantics boils down to find an equivalent mapping between output formulas. Note that due to obfuscation such as register renaming, basic blocks could use different registers or variables to provide the same functionality. As a result, current approaches exhaustively try all possible pairs to find if there exists a bijective mapping between output formulas. Fig. 3 shows such formulas mapping attempt for the output formulas shown in Fig. 2. The variables shown in the leftmost column are from basic block 1 in Fig. 2; while the variables in the upmost row are from basic block 2. The "true" or "false" indicates the result of equivalence checking, such as whether $edx\_1 = eax\_3$. After 10 times comparisons, we identify a perfect matched permutation and therefore conclude that these two basic blocks are truly equivalent.

Based on the matched basic blocks, BinHunt [14] computes the similarity of control flow graphs of two binaries by graph isomorphism. The follow-up work, iBinHunt [23], finds semantic differences between execution traces and utilizes multi-tag taint analysis to reduce the number of basic block matches. Luo et al. [22] detect software plagiarism by matching the longest common subsequence of semantically equivalent basic blocks.

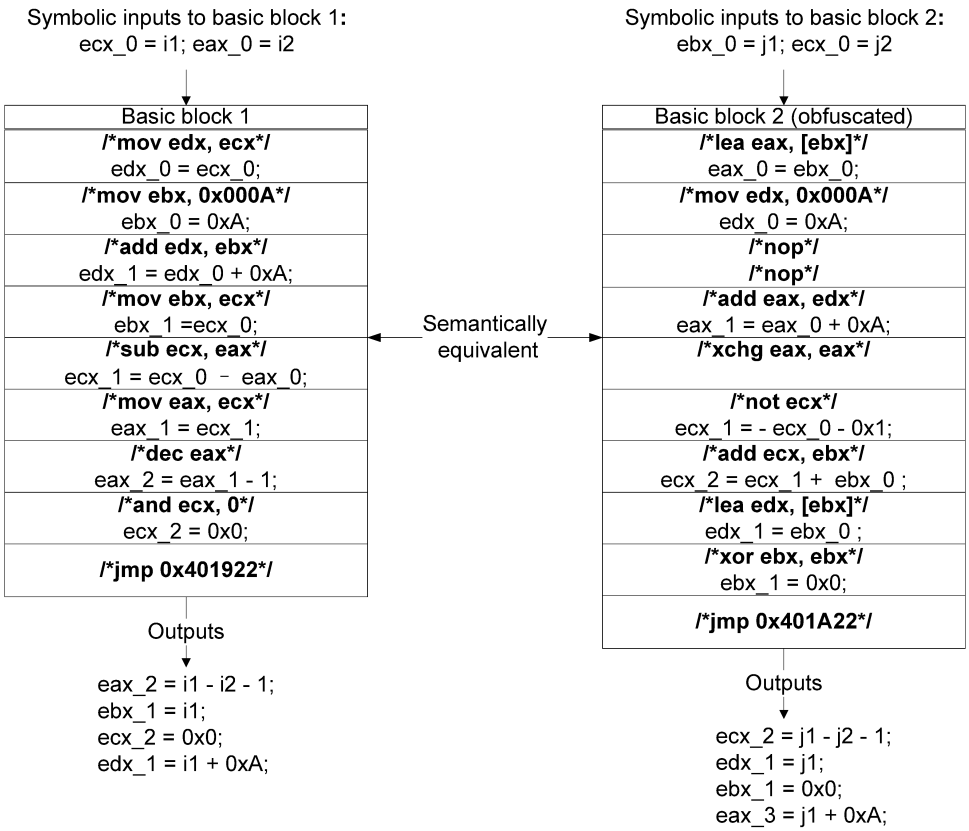


Fig. 2 Example: basic block symbolic execution. The symbolic execution is performed based on IL (for brevity, we do not show the modification to the EFLAGS bits).

Query result	eax_3	ebx_1	ecx_2	edx_1
eax_2	false		**true**	false
ebx_1	false		false	**true**
ecx_2		**constant (0)**		
edx_1	**true**		false	false

Fig. 3 Output formulas equivalence query results.

3 Performance Bottleneck

We look into the overhead imposed by semantics-based binary diffing and find that there are two factors dominating the cost. The first is the high number of invocations of constraint solver. Recall that current approaches check all possible permutations of output formulas mapping. The constraint solver will be invoked every time when verifying the equivalence of formulas. For example, two basic blocks in Fig. 2 have three symbolic formulas and one constant value respectively. As shown in Fig. 3, We have to employ constraint solver at most nine times to find an equivalent mapping between the three output formulas. Too frequently calling constraint solver incurs a significant performance penalty. The second is the slow processing speed of symbolic execution. Typically symbolic execution is much slower than native execution, because it simulates each x86 instruction by interpreting a sequence of IL statements.

To quantitatively study such performance bottleneck, we select 5 malware families from our evaluation dataset (see Section 5.2): four families have a large number of samples (StartPage, Delf, Mimail and NGVCK), and one family (Ping) has the maximal code size. We apply iBinHunt [23] to perform pairwise comparison within each family. The constraint solver we used is STP [13]. As shown in Fig. 4, we divide the overall processing time into three parts: constraint solver solving time ("STP" bar), symbolic execution time ("SE" bar), and other operations ("Others" bar). Apparently,

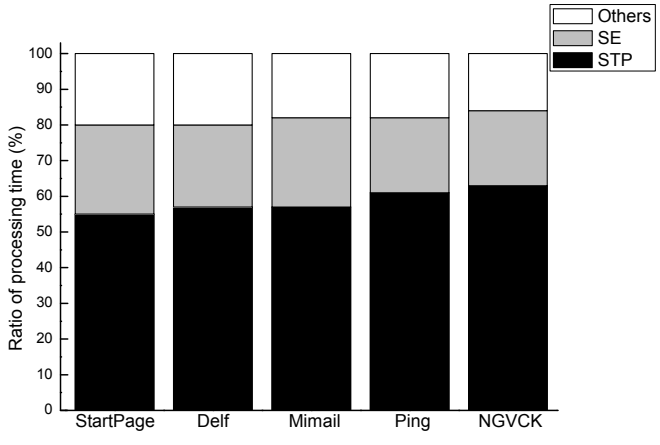


Fig. 4 Ratio of processing time of iBinHunt.

STP's processing time accounts for most of running time of iBinHunt (more than 50%). Note that the experiments on EXE [6] and KLEE [5] report similar results, in which running time is dominated by the constraint solving. Besides, the symbolic execution also takes up to about 23% running time. Thus, an immediate optimization goal is to mitigate too frequent invocations of constraint solver and slow symbolic execution.

4 MalwareHunt Design

When we compare malware variants to identify their relationships (a.k.a, lineage inference [16]), our key observation is that similar malware variants are likely to share common code [20]. For example, all of the Email-Worm.Win32.NetSky samples in our dataset search for email addresses on the infected computer and use SMTP to send themselves as attachments to these addresses. The net result is we have to re-compare a large number of basic blocks that have been previously analyzed. Therefore, our key method is to utilize memoization optimization to reuse previous computed results. To this end, we first normalize basic blocks to reverse some obfuscation effects. Then, MD5 value of the byte sequence of each basic block is calculated. After that, 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. Besides, we also concretize symbolic formulas and cache equivalence queries to further cut down the invocation numbers of constraint solver. Next we will discuss each step in detail.

4.1 Normalization

The comparison unit of most semantics-based binary diffing work is basic block [14,23,22]. However, several obfuscation methods can split a single basic block into multiple ones. As a result, too much extra basic block comparisons will take up computing resources. We first perform normalization to reverse such obfuscation effects. Currently, we consider two major obfuscation methods: instruction reordering and opaque predicate obfuscation. The example of instruction reordering is shown in Fig. 1(a), in which the new basic blocks are connected through direct jump (e.g., `jump loc_0031`). Reversing instruction reordering is straightforward. We merge all the basic blocks that have only one predecessor and one successor. For example, the new generated basic blocks in Fig. 1(a) will be merged into a single basic block again.

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 afterwards. For example, predicate ($x^3 - x \equiv 0 \pmod 3$) in Fig. 6 is true for all integers x. Opaque predicates have been widely used to introduce redundant branches for the purpose of control flow obfuscation [24]. To handle opaque predicates, we rely on recent work on logic-oriented opaque predicate detection [24]. 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 Fig. 6, the unreachable paths and redundant predicates will be discarded; the basic blocks split by the opaque predicate will be merged together.

In addition, we also normalize basic blocks to ignore offsets that may change due to code relocation and some `nop` instructions. Binary 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 calculation of hash value. The purpose of normalization is to ignore such effects and make the hash value more general. We preform the normalization on the intermediate language (IL). The RISC-like intermediate language and static single assignment (SSA) format of IL are convenient for our processing, and also represent many functionally equivalent instructions (e.g., `xor eax, eax` and `and eax, 0`) in the same way. Taken the basic block 2 in Fig. 2 as an example, Fig. 5 shows that we normalize the basic block by replacing address values with zeros and remove all `nop` statements. Then we calculate the MD5 value of the basic block's byte

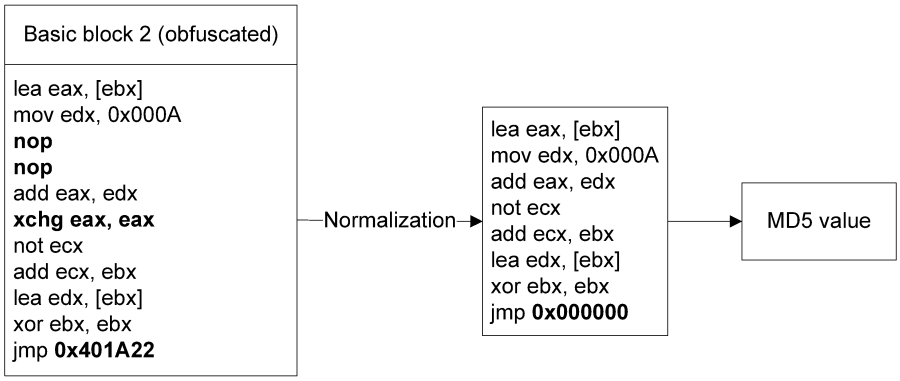


Fig. 5 Basic block normalization. The normalization is performed on the intermediate language. Here we only show assembly code for the sake of brevity.

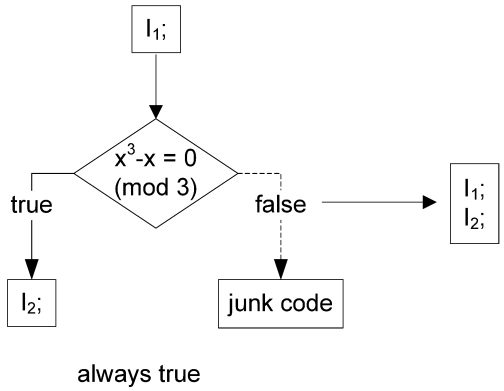


Fig. 6 Deobfuscate opaque predicate.

sequence, which will be used in the union-find set operations.

4.2 Union-Find Set of Equivalent Basic Blocks

Our first optimization is to utilize union-find set [8], an efficient tree-based data structure, to reuse previous matched equivalent basic blocks. We define the three major operations of union-find set as follows.

1. `MakeSet`: Create an initial subset structure containing one element, which is represented by a basic block's MD5 value. Each element's parent points to itself and has 0 depth.
2. `Find`: Determine which subset a basic block belongs to. `Find` operation is used to find two basic blocks are equivalent if both of them are within the same subset.
3. `Union`: Unite two subsets into a new single subset. The depth of new set will be updated accordingly.

The elements within a subset build up a tree structure. `Find` operation will always recursively traverse the tree structure. However, the tree structure might degrade to a long list of nodes, which incurs $\mathcal{O}$(n) time in the worst case for `Find` operation. To avoid highly unbalanced searching tree, an improved path compression and weighted union algorithm are applied to speed up `Find` operation. Algorithm 1 shows the pseudo-code of `MakeSet`, `Find` and `Union`. `MakeSet` creates an initial set containing only one basic block. Path compression is a way to flatten the structure of the tree when `Find` recursively explores on it. As a result, each node's parent points to the root `Find` returns (Line 7). Weighted `Union` algorithm attaches the tree with smaller depth to the root of taller tree (Line 17, Line 20), which only increases depth when depths are equal (Line 24).

Fig. 7 shows an example of maintaining an union-find set. Given previously matched basic block pairs

Algorithm 1 MakeSet, Find and Union

```
1: function MakeSet(a)    // a represents a basic block
2:     a.parent ← a
3:     a.depth ← 0
4: end function
5: function Find(a)    // path compression
6:     if a.parent ≠ a then
7:         a.parent ← Find(a.parent)
8:     end if
9:     return a.parent
10: end function
11: function Union(a,b)    // weighted union
12:     aRoot ← Find(a)
13:     bRoot ← Find(b)
14:     if aRoot = bRoot then
15:         return
16:     end if
17:     if aRoot.depth < bRoot.depth then
18:         aRoot.parent ← bRoot
19:     else
20:         if aRoot.depth > bRoot.depth then
21:             bRoot.parent ← aRoot
22:         else
23:             bRoot.parent ← aRoot
24:             aRoot.depth ← aRoot.depth + 1
25:         end if
26:     end if
27: end function
```

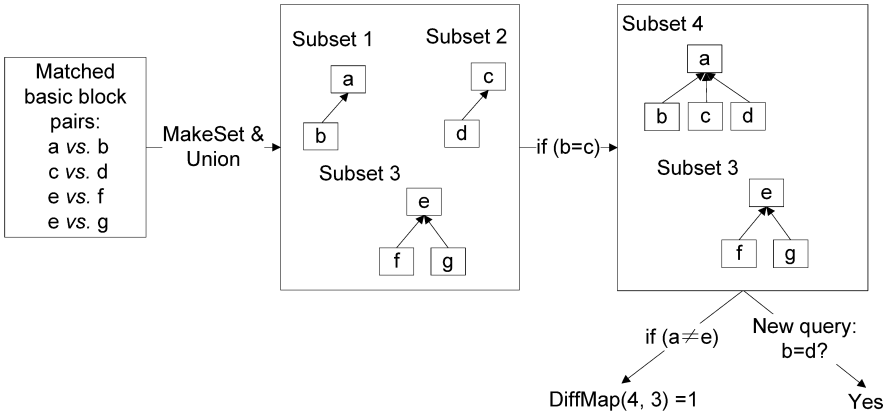


Fig. 7 Example of MakeSet-Union-Find operations.

(as shown in left most block), after initial `MakeSet` and `Union` operations, we get three subsets, that is, $\{a, b\}$, $\{c, d\}$ and $\{e, f, g\}$. Then assuming b and c, two basic blocks coming from different subsets (subset 1 and 2), have the same semantics, that means all of the basic blocks in these two subsets are in fact equivalent. Therefore, we perform weighed union and path compression to join the two subsets to a new subset (subset 4). The resulting tree is much flatter with a depth 1. After the union, we can immediately determine that b and d are equivalent, even if these two basic blocks were not compared before. 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. As shown in the lower right side of Fig. 7, if we find out a and e are different, we can safely conclude that basic blocks in subset 4 are not equivalent to the ones in subset 3, without the need for comparing them pair-by-pair anymore.

4.3 Concretize Symbolic Formulas and Cache Equivalence Queries

Fig. 3 shows a drawback of semantics-based binary diffing: without knowing the mapping of output formulas for equivalence checking, current approaches have to exhaustively try all possible permutations. To ameliorate this issue, we introduce a sound heuristic that *if two basic block output formulas are equivalent, they should generate equal values when substituting symbols with the same concrete value.* Therefore, we give preference to the symbolic formulas producing the same value after concretization. Taken the output formulas in Fig. 2 as an example, we substitute all the input symbols with a single concrete value 1. In this way, we can quickly identify the possible mapping pairs, and then we verify them again with STP. As a result, we only invoke STP 3 times, instead of 9 times as is previously done. Note that using STP for double-checking is necessary, as two symbolic formulas may happen to generate the same value. For example, $i << 1$ is equal to $i * i$ when $i = 2$.

Besides, in order to further reduce the invocations of STP when possible, we manage a `QueryMap` to cache the result of equivalence queries, which is quite similar to constraints caching adopted by EXE [6] and KLEE [5]. The key of `QueryMap` is MD5 value of an equivalence query, such as whether `edx_1 = eax_3` in Fig. 2; the value of `QueryMap` stores STP query result (true or false). Before calling STP on a query, we first check `QueryMap` to see whether it gets a hit. If not, we'll create a new `(key, value)` entry into QueryMap after we verify this query with STP.

4.4 Basic Blocks Fast Matching

We merge all three optimization methods discussed above together to comprise our *basic blocks fast matching* algorithm (as listed in Algorithm 2). Our basic blocks fast matching exploits syntactical information and previous result for early pruning. When comparing two basic blocks, we first normalize the basic blocks 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. At the same time, we leverage heuristic of concretizing symbolic formulas and QueryMap cache to reduce the invocations of

Algorithm 2 Basic Block Fast Matching

```
    a, b : two basic blocks to be compared
 1: function FastMatching(a, b)
 2:     a' ← Normalize(a)
 3:     b' ← Normalize(b)
 4:     if Hash(a') = Hash(b') then
 5:         return True
 6:     end if
 7:     if Find(a') = Find(b') then // within the same subset
 8:         return True
 9:     end if
10:     if DiffMap(Find(a'), Find(b'))=1 then // semantically different subsets
11:         return False
12:     else
13:         Perform symbolic execution on a' and b'
14:         Check semantical equivalence of a' and b'
15:         if a' ∼ b' then // a', b' are semantically equivalent
16:             Union(a', b')
17:             Update DiffMap
18:             return True
19:         else   // a', b' are not semantically equivalent
20:             Set DiffMap(Find(a'), Find(b'))
21:             return False
22:         end if
23:     end if
24: end function
```

STP. After that we update union-find set and DiffMap accordingly (Line 15∼22).

Although the attempt to find all the matched basic blocks is equal to solving the halting problem [19], the resulting union-find sets have interesting application on the practical point of view. For example, a mutation insensitive signature [32] can be generated to capture possible metamorphic variants and malware finegrained relationship information can even be recovered.

5 Experimental Evaluation

We perform our experiments with several objectives in mind. First, we want to evaluate the effectiveness of MalwareHunt in the task of malware lineage inference. Also, we are interested in the effect of our basic blocks fast matching over time. Second, we want to study the effect of our approach to alleviating the semantics-based binary diffing performance bottleneck. At last, we present the detailed optimization breakdown.

5.1 Implementation

We have implemented the idea of MalwareHunt on top of iBinHunt [23], a binary diffing tool to find semantic differences between execution traces, with 1,820 OCaml lines of code. Fig. 8 shows the architecture of MalwareHunt and the newly added components. We preform the basic block normalization on the Vine IL [35], and the theorem prover we used is STP [13]. 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. Also, we write 500 lines of Perl scripts to glue all components together to automate the comparison process.

5.2 Experiment Setup

We collected malware samples from VX Heavens[1] and leveraged an online malware scan service, VirusTotal[2], to classify the samples into an initial 15 families. These malware samples range from metamorphic virus to considerably large Trojan horse. The dataset statistics is shown in Table 1. The experimental data are collected during malware lineage inference within each family, that is, we perform a pairwise comparison to determine relationships among malware variants. The forth column of Table 1 lists the number of pairwise comparison of each family and the total number is 1,664. Our testbed consists of Intel Core i7-3770 processor (Quad Core with 3.40GHz) and 8GB memory. The malware execution traces are collected when running in Temu [39], a whole-system emulator. Since most of malware samples are packed, we employ a generic unpacking plug-in [17] to monitor malware sample's unpacking and start to record trace only when the execution reaches the original entry point (OEP) [21].

There are three kinds of metamorphic virus families: Lexotan32, MetaPHOR, and NGVCK. Lexotan32 and MetaPHOR are self-mutating malware; that is, they embed the metamorphic engine within the virus body [36]. We select 20 copies of Cygwin[3] utility `bzip2` as the "goat" binaries to be infected by both Lexotan32 and MetaPHOR. Since these self-mutating viruses do not mutate their host code, choosing the same copies of goat files can help us identify the morphing code. During our evaluation, the running goat executables will infect themselves iteratively, and each infection will yield a new generation variant. For NGVCK (next generation virus creation kit) [37], its engine is separated from the malicious body. We generate 24 NGVCK virus variants in terms of assembly source code. Then, we use TASM 5.0 Assembler to compile the source code into binary.

[1] http://vxheaven.org/src.php
[2] https://www.virustotal.com/
[3] https://www.cygwin.com

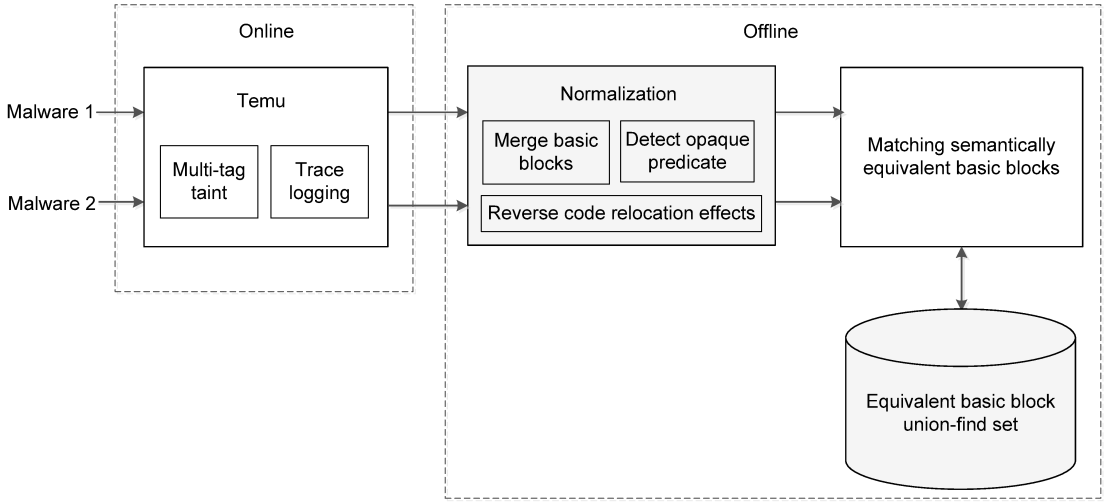


Fig. 8 The architecture of MalwareHunt. The grey components are newly added on top of iBinHunt [23].

Table 1 Dataset statistics

Malware Family	Category	#Samples	#Comparison	Size(kb)/Std.Dev.
Dler	Trojan	10	45	28/6
StartPage	Trojan	21	210	10/1
Delf	Trojan	24	276	17/4
Ping	Backdoor	8	28	247/41
SpyBoter	Backdoor	16	120	34/16
Progenic	Backdoor	6	15	88/27
Bube	Virus	10	45	12/7
Lexotan32	Virus	20	190	64/15
MetaPHOR	Virus	20	190	65/20
NGVCK	Virus	24	276	150/35
MyPics	Worm	12	66	31/4
Bagle	Worm	9	36	40/17
Mimail	Worm	17	136	17/6
NetSky	Worm	7	21	41/12
Sasser	Worm	5	10	60/28

5.3 Malware Lineage Inference Performance

We first quantify the effects of the set of optimizations we presented in our basic blocks fast matching algorithm (Algorithm 2). Fig. 9 shows the speedup of malware lineage inference for each family when applying the optimizations cumulatively on MalwareHunt. Our baseline for this experiment is a conventional MalwareHunt without using any optimization we proposed. The "O1" bar indicates the effect of normalization, which can quickly identify basic block pairs with the same byte sequences after our normalization. The effect of normalization is remarkable on several families such as Ping and Bube, in which instructions are quite similar in syntax. Recall that our normalization reverses the effect of instruction reordering and opaque predicate, which are commonly used code mutation methods. As a result, we also achieve a notable improvement on comparing metamorphic malware variants (Lexotan32, MetaPHOR, and NGVCK).

The "O2" bar captures the effect of the union-find set and DiffMap, which record previously compared results. The optimization O2 results in a significant speedup from 1.4X to 2.9X on average. Especially for some highly obfuscated malware families, such as Delf and Bagle, O2 outperforms O1 by a factor of up to 3.1. The "O3" bar, denoting concretizing symbolic formulas, introduces an improvement by 17% on average. The optimization of QueryMap (O4) offers an enhanced performance improvement by average 30% and with a peak value 46% for NetSky. Particularly, since StartPage samples adopt different implementation ways to tamper with the startup page of IE browsers, we observe large similarity distances among StartPage variants. In spite of this, our approach still accelerates the malware lineage inference greatly.

In addition, we study the effect of our basic blocks fast matching over time as well. We choose Sasser to test because the impact of the optimizations on Sasser is close to the average value. As shown in Fig. 10, as

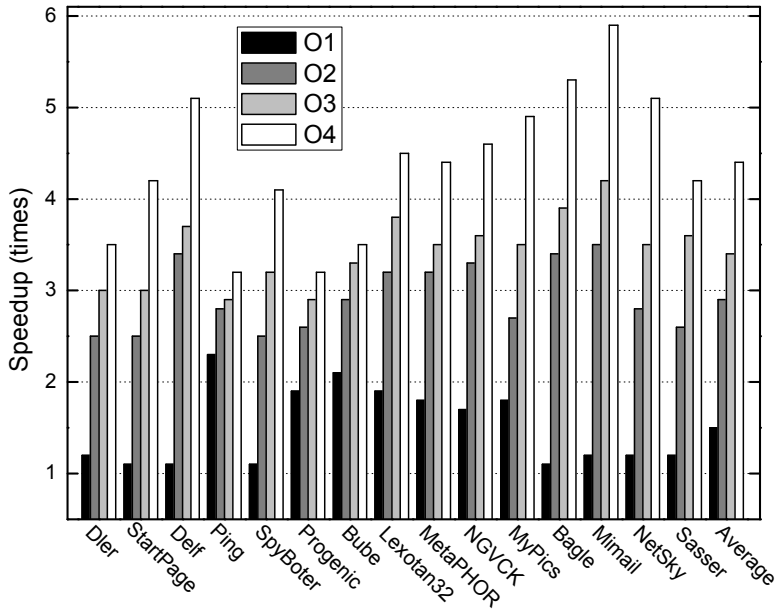


Fig. 9 The impact of basic blocks fast matching on malware lineage inference: O1 (normalization), O2 (O1 + union-find set and DiffMap), O3 (O2 + concretizing symbolic formulas), O4 (O3 + QueryMap).

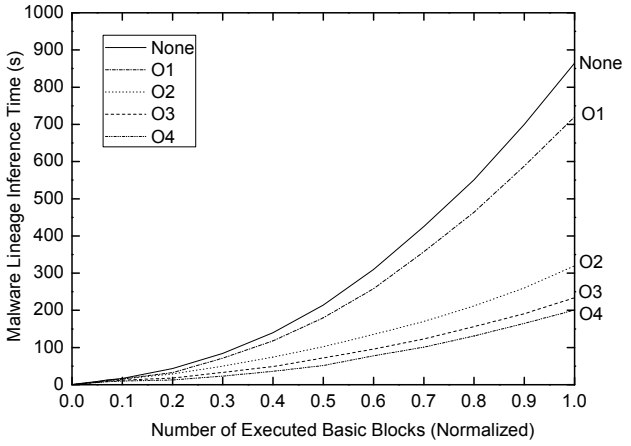


Fig. 10 The effect of our optimizations over time on Sasser family.

the union-find set and QueryMap are enlarged, our approach becomes more effective over time. The number of executed basic blocks is normalized so that data can be collected across intra-family comparisons.

5.4 Alleviate Performance Bottleneck

In Section 3, we have identified two factors that dominate the cost of semantics-based binary diffing: namely symbolic execution and constraint solver. In this experiment, we study the effect of our optimizations on these two performance bottlenecks. The column 2∼4 of Table 2 lists the average symbolic execution time and speedup before/after optimization when comparing two malware variants in each family using MalwareHunt. Similarly, the column 5∼7 shows the effect to reduce the number of STP invocations. In summary, our approach outperforms conventional MalwareHunt in terms of less symbolic execution time by a factor of 4.1x on average, and fewer STP invocations by 4.5x on average.

5.5 Optimization Breakdown

Table 3 presents our optimization breakdown when performing lineage inference for the five large malware families shown in Fig. 4. The first row shows the ratio of matched basic block pairs with the same byte sequences after normalization (line 4 in Algorithm 2). The relatively small ratio also indicates the necessity of semantics-based binary diffing approach. The next two rows list statistics of the union-find set, including the number of union-find subsets and the maximum number of items in one subset. As the key property of metamorphic mutation is semantics-preserving, NGVCK has the maximum number (16) of equivalent basic blocks. The row 4 and 5 present the hit rate of union-find set (line 7 in Algorithm 2) and DiffMap (line 10 in Algorithm 2). The row 6 shows the time cost incurred by

Table 2 Improvement to symbolic execution and STP invocations.

Malware Family	SE Times (s)			# STP Invocations		
	None	Optimization	Speedup	None	Optimization	Speedup
Dler	10.1	2.5	4.0	1,123	346	3.2
StartPage	13.5	3.3	4.1	1,350	314	4.3
Delf	8.8	2.0	4.4	1,685	324	5.2
Ping	32.2	10.7	3.0	6,740	1,926	3.5
SpyBoter	16.6	4.4	3.8	2,020	493	4.1
Progenic	20.2	6.3	3.2	2,566	856	3.0
Bube	5.1	1.8	2.8	8,47	250	3.4
Lexotan32	12.7	2.9	4.3	1,278	228	5.6
MetaPHOR	16.2	3.6	4.5	1,394	278	5.0
NGVCK	25.6	4.8	5.3	5,458	1,186	4.6
MyPics	11.4	2.5	4.6	1,235	257	4.8
Bagle	24.4	5.1	4.8	5,570	1,092	5.1
Mimail	9.0	1.7	5.3	2,901	484	6.0
NetSky	20.6	4.6	4.5	4,958	972	5.1
Sasser	24.2	6.2	3.9	5,616	1,338	4.2
Average			4.1			4.5

Table 3 Optimization breakdown.

	Optimization breakdown	StartPage	Delf	Mimail	Ping	NGVCK
1	Normalization ratio	9%	12%	12%	51%	34%
2	# union-find subsets	125	130	304	546	346
3	Max. # basic blocks in one subset	5	6	8	10	16
4	union-find hit rate	36%	44%	37%	41%	52%
5	DiffMap hit rate	47%	53%	44%	54%	38%
6	Union-find set and DiffMap cost (s)	9.5	14.3	12.0	13.7	12.2
7	QueryMap hit rate	62%	70%	65%	75%	56%
8	QueryMap cost (s)	8.6	8.8	6.4	7.6	6.8
9	Concretizing saving	60%	65%	55%	52%	46%
10	Memory cost (MB)	10	12	28	45	36

building and managing the union-find set structure and DiffMap. The following two rows lists hit rate and time cost for QueryMap. The saving of concretizing symbolic formulas is shown in row 9, in which we avoid at least 46% output variables comparisons. At last, we present the overall memory cost to maintain union-find set, DiffMap and QueryMap. Reassuringly, compared to the performance improvement, the overhead introduced by our optimization is small.

6 Related Work

In this section, we first present previous work on binary diffing, which is the most related to MalwareHunt in spirit. Then we introduce literature on symbolic execution optimization, which inspires our approach to basic block memoization.

6.1 Binary Diffing

Hunting differences between two binaries (a.k.a, binary diffing) has a wide variety of applications in software security area, such as exploits or bugs exploration [12, 1,31], reverse engineering [14,23,11] and code reuse detection [27,22]. The recent work [9] defines a formal semantic model for binary diffing. The previous work that relies on control flow graph or instruction fingerprint hashing to compare binaries [12,1,28] can be evaded by sophisticated obfuscation methods. Our efforts attempt to speed up semantics-based binary diffing, which can find equivalent binary pairs that reveal syntactic differences [14,22,23,27]. We have introduced the latest work in this direction in Section 2. The most relevant binary diffing method to MalwareHunt is iBinhunt [23]. We are all trace-oriented binary diffing tools to match basic block pairs and also utilize multi-tag taint analysis to reduce the number of possible matches. However, MalwareHunt is designed to compare a large number of obfuscated malware variants. Compared to iBinhunt, MalwareHunt is augmented with better resilience to various code obfuscation methods (e.g. opaque predicate) and a set of memoization optimization methods. As a result, MalwareHunt has a better accuracy and performance.

6.2 Symbolic Execution Optimization

A set of our memoization optimization methods to speed up semantics-based binary diffing are inspired by symbolic execution optimization work. Yang et al. [38] proposed `Memoise`, a trie-based data structure to cache the key elements of symbolic execution, so that successive forward symbolic execution can reuse previously computed results. Our union-find structure is like Memoise in that we both maintain an efficient tree-based data structure to avoid re-computation. However, our approach aims to accelerate basic blocks matching, and our symbolic execution is limited in a basic block, which is a straight-line code without path conditions. Our optimization of caching equivalence queries is inspired by both EXE [6] and KLEE [5], which cache the result of path constraint solutions to avoid redundant constraint solver calling. Different from the complicated path conditions cached by EXE and KLEE, our equivalence queries are simple and compact. As a result, our QueryMap has a higher cache hit rate. Malware normalization relies on ad-hoc rules to undo the obfuscations applied by malware developers [4, 7]. Our normalization mainly focuses on the obfuscation methods that may change the structure of a basic block, namely instruction reordering and opaque predicate. Besides, we also eliminate the effect of memory relocation and instruction idioms.

7 Conclusion

The best way to reconcile the scalability issue of symbolic execution and its applications is an active research topic. The high-performance penalty introduced by the state-of-the-art semantics-based binary diffing approaches restricts their application from large scale application such as analyzing a large number of malware samples. In this paper, we first study the cruxes leading to the performance bottleneck and then proposed normalized basic block memoization optimization to speed up semantics-based binary diffing. Our approach consists of basic block normalization, maintaining a union-find set structure, concretizing symbolic formulas and caching equivalence queries. The experiment on malware lineage inference demonstrated the efficacy of our optimizations with only minimal overhead. Although we evaluated our approach with the application to malware analysis, our basic blocks fast matching solution can be seamlessly weaved into other binary diffing approaches based on equivalent basic blocks.

Acknowledgements

This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grants N00014-13-1-0175 and N00014-16-1-2265.

References

1. Martial Bourquin, Andy King, and Edward Robbins. Binslayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW '13)*, 2013.
2. David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposimu on Security and Privacy (SP'08)*, 2008.
3. D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'06)*, 2006.
4. Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of the International Symposium of Secure Software Engineering*, 2006.
5. Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 2008 USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.
6. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 2006 ACM Conference on Computer and Communications Security (CCS'06)*, 2006.
7. Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005.
8. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Second ed.)*, chapter 21: Data structures for Disjoint Sets, pages 498–524. MIT Press, 2001.
9. Mila Dalla Preda, Roberto Giacobazzi, Arun Lakhotia, and Isabella Mastroeni. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, 2015.
10. The Mental Driller. Metamorphism in practice or How I made MetaPHOR and what I've learnt. `http://vxheaven.org/lib/vmd01.html`, last reviewed, 04/14/2015.
11. Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security'14)*, 2014.
12. Halvar Flake. Structural comparison of executable objects. In *Proceedings of the 2004 GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'04)*, 2004.
13. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 2007 International Conference in Computer Aided Verification (CAV'07)*, 2007.

14. Debin Gao, Michael K. Reiter, and Dawn Song. BinHunt: Automatically finding semantic differences in binary programs. In *Poceedings of the 10th International Conference on Information and Communications Security (ICICS'08)*, 2008.
15. Matthias Jacob, Mariusz H. Jakubowski, Prasad Naldurg, Chit Wei Saw, and Ramarathnam Venkatesan. The superdiversifier: Peephole individualization for software protection. In *Proceedings of the 3rd International Workshop on Security (IWSEC'08)*, 2008.
16. Jiyong Jang, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security'13)*, 2013.
17. Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM '07)*, 2007.
18. J. Zico Kolter and Marcus A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the 10th ACM SIGKDD conference (KDD'04)*, 2004.
19. Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13)*, 2013.
20. Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, 2012.
21. Limin Liu, Jiang Ming, Zhi Wang, Debin Gao, and Chunfu Jia. Denial-of-service attacks on host-based generic unpackers. In *Proceedings of the 11th International Conference on Information and Communications Security (ICICS'09)*, 2009.
22. Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, 2014.
23. Jiang Ming, Meng Pan, and Debin Gao. iBinHunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)*, 2012.
24. Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. LOOP: Logic-oriented opaque predicates detection in obfuscated binary code. In *Proceedings of the 22nd ACM Conference on Computer and Communications Securit (CCS'15)*, 2015.
25. Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized semantics-based binary diffing with application to malware lineage inference. In *Proceedings of the 30th IFIP International Information Security and Privacy Conference (IFIP SEC'15)*, 2015.
26. Beng Heng Ng, Xin Hu, and Atul Prakash. A study on latent vulnerabilities. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS'10)*, 2010.
27. Beng Heng Ng and Atul Prakash. Exposé: Discovering potential binary code re-use. In *Proceedings of the 37th IEEE Annual Computer Software and Applications Conference (COMPSAC'13)*, 2013.
28. Jeong Wook Oh. DarunGrim: A patch analysis and binary diffing too. `http://www.darungrim.org/`, last reviewed, 10/26/2015.
29. Orr. The Molecular Virology of Lexotan32: Metamorphism Illustrated. `http://www.openrce.org/articles/full_view/29`, last reviewed, 04/14/2015.
30. Panda Security. Annual report 2013 summary. `http://press.pandasecurity.com/wp-content/uploads/2010/05/PandaLabs-Annual-Report_2013.pdf`, last reviewed, 10/25/2015.
31. Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.
32. Mila Dalla Preda. The grand challenge in metamorphic analysis. In *Proceedings of the 6th International Conference on Information Systems, Technology and Management (ICISTM12)*, 2012.
33. Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys*, 46(1), 2013.
34. Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, February 2012.
35. Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*, 2008.
36. Sudarshan Madenur Sridhara and Mark Stamp. Metamorphic worm that carries its own morphing engine. *Computer Virology*, 9(2):49–58, 2013.
37. Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Computer Virology*, 2(3):211–229, 2006.
38. Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*, 2012.
39. Heng Yin and Dawn Song. TEMU: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.