Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference

Jiang Ming, Dongpeng Xu, Dinghao Wu

Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference

Jiang Ming, Dongpeng Xu, and Dinghao Wu

The Pennsylvania State University
University Park, PA 16802, U.S.A.
{jum310, dux103, dwu}@ist.psu.edu

Abstract. Identifying differences between two executable binaries (binary diffing) has compelling security applications, such as software vulnerability exploration, "1-day" exploit generation and software plagiarism detection. Recently, binary diffing based on symbolic execution and constraint solver has been proposed to look for the code pairs with the same semantics, even though they are ostensibly different in syntactics. Such logical-based method captures intrinsic differences of binary code, making it a natural choice to analyze highly-obfuscated malicious program. However, semantics-based binary diffing suffers from significant performance slowdown, hindering it from analyzing large-scale 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 *basic blocks fast matching* 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 purpose to concretize symbolic formulas and cache equivalence queries to further cut down the invocation times of constraint solver. We have implemented our technique on top of iBinHunt and evaluated it on 12 malware families with respect to the performance improvement when performing intra-family comparisons. Our experimental results show that our methods can accelerate symbolic execution from 2.8x to 5.3x (with an average 4.0x), and reduce constraint solver invocation by a factor of 3.0x to 6.0x (with an average 4.3x).

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 [16], automatic "1-day" exploit generation [1] and software plagiarism detection [14]. Conventional approaches can quickly locate syntactical differences by measuring instruction sequences [20] or byte N-grams [11]. However, such syntax-based comparison can be easily defeated by various obfuscation techniques, such as instruction substitution [9], binary packing [19] and self-modifying code [2]. The latest binary diffing approaches [8, 15] 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. Then the equivalence of formulas are verified by a constraint solver. Such logic-based comparison, capturing the intrinsic semantic differences, has been applied to finding differences of program versions [8], comparing inter-procedural control flows [15] and identifying code reuse [14, 17].

On the other hand, malware authors frequently update their malicious code to circumvent security countermeasures. According to the latest annual report of Panda Security labs [18], 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 severely restricted its application in large scale analysis, such as malware lineage inference [10], which normally requires pair-wise comparison to identify relationships among malware variants. 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 both factors, we propose *basic blocks fast matching* by reusing previously compared results, which consists of three optimization methods to accelerate equivalent basic block matching. Our key insight is that malware variants are likely to share common code [12]; new variant may be just protected with a different packer or incremental updates. As a result, we exploit code similarity by adopting union-find set [6], an efficient tree-based data structure, to record semantically equivalent basic blocks which have already been identified. Essentially, the union-find structure stores the md5 value of each matched basic block after normalization. Maintaining the union-find structure during successive comparisons allows direct reuse of previous results, without the need for re-comparing them. Moreover, to further cut down the high invocation times of constraint solver, we purpose to concretize symbolic formulas and cache equivalence queries. We have implemented these optimizations on top of iBinHunt [15] and evaluated them when performing malware lineage inference on 12 malware families. Our experimental results show that our methods can speed up malware lineage inference, symbolic execution and constraint solver by a factor of 4.4x, 4.0x and 4.3x, respectively. Our proposed solution focuses on accelerating basic blocks matching and therefore can be seamlessly woven into other binary diffing approaches based on equivalent basic blocks. 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 *basic blocks fast matching* to enable more efficient binary comparison, including maintaining a union-find set structure, concretizing symbolic formulas and caching equivalence queries.
3. We implement our approach on a state-of-the-art binary diffing tool and demonstrate its efficacy in malware lineage inference.

The rest of the paper is organized as follows. Section 2 provides the background information. Section 3 studies the performance bottleneck of semantics-

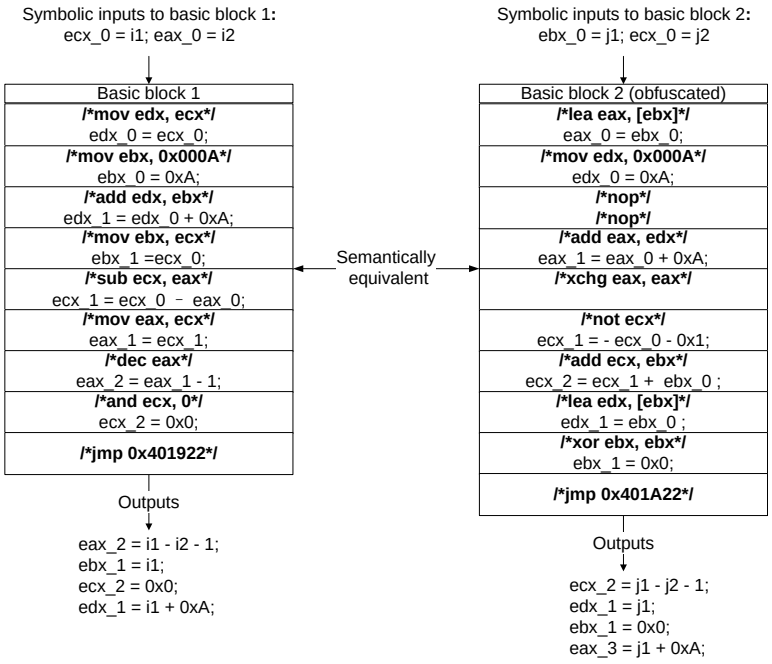

Fig. 1. Basic block symbolic execution

based binary diffing. Section 4 describes our optimization methods in detail. We evaluate our approach 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. The core method of current approaches [8, 15, 14] are matching semantically equivalent basic blocks. Basic block is a straight line code with only one entry point and only one exit point, which makes a basic block highly amenable to symbolic execution (e.g., without conjunction of path conditions). Fig. 1 presents a motivating example to illustrate how semantics of a basic block is simulated by symbolic execution. The two basic blocks in Fig. 1 are semantically equivalent, even though they have different x86 instructions (listed in bold). In practice, symbolic execution is performed on a RISC-like intermediate language (IL), which represents complicated x86 instructions as simple Single Static Assignment (SSA) style statements (e.g., ecx_0, edx_1).

Taken the inputs to the basic block as symbols, 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. Hence current approaches exhaustively try all possible pairs to find if there exists a bijective mapping

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. 2. Output formulas equivalence query results

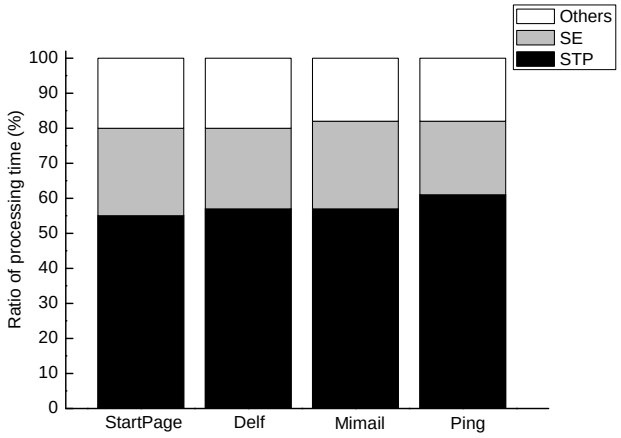

Fig. 3. Ratio of processing time of iBinHunt

between output formulas. Fig. 2 shows such formulas mapping attempt for the output formulas shown in Fig. 1. 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 [8] computes the similarity of control flow graphs of two binaries by graph isomorphism; iBinHunt [15] finds semantic differences between execution traces and Luo et al. [14] detect software plagiarism by matching "longest common subsequence of semantically equivalent basic blocks".

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. 1 have 3 symbolic formulas and 1 constant value respectively. As shown in Fig. 2, We have to employ constraint solver at most 9 times to find an equivalent mapping between 3 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 selected 4 malware families from our evaluation dataset (see Section 5.1): 3 families have large number of samples (StartPage, Delf and Mimail) and one family (Ping) has the maximal code size. We applied iBinHunt [15] to compare execution traces of pair-wise samples within each family. The constraint solver we used is STP [7]. Fig. 3 shows the ratio of each stage's processing time on average: constraint solver solving time ("STP" bar), symbolic execution time ("SE" bar) and other operations ("Others" bar). Apparently, STP's processing time accounts for most of running time of iBinHunt (more than 50%). Experiments on EXE [4] and KLEE [3] 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 Optimization

4.1 Union-Find Set of Equivalent Basic Blocks

When we compare malware variants to identify their relationships (a.k.a, lineage inference [10]), our key observation is that similar malware variants are likely to share common code [12]. 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 large number of basic blocks that have been previously analyzed. Therefore our first optimization is to utilize union-find set [6], an efficient tree-based data structure, to reuse previous matched equivalent basic blocks. More specifically, we first normalize basic blocks to ignore offsets that may change due to code relocation and some `nop` instructions. MD5 value of the byte sequence of each basic block is then calculated. Secondly, 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. Next we'll discuss these two steps in detail.

Normalization 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. Taken the basic block 2 in Fig. 1 as an example, Fig. 4 presents how to normalize a basic block, in which we replace address values with zeros and remove all `nop` statements. After that, the MD5 value of the basic block's byte sequence is calculated.

Maintain Union-Find Set We define the three major operations of union-find set as follows:

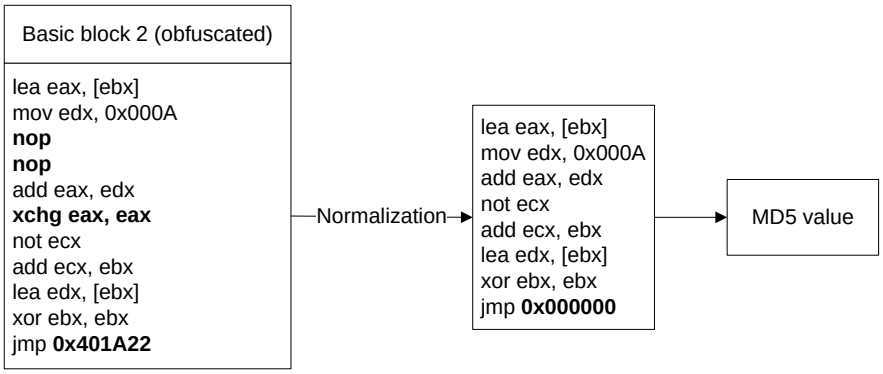

Fig. 4. Basic block normalization

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 can be used to quickly 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 on 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`. 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. 5 shows an example of maintaining an union-find set. Given previously matched basic block pairs (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 union, we can immediately determine that b and d are equivalent, even if these two basic blocks were not compared before. In addition to union-fine 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. 5, 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 anymore.

Algorithm 1 MakeSet, Find and Union

```
1: function MakeSet(a)                                   ▷ a is 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
```

4.2 Concretizing Symbolic Formulas

Fig. 2 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 symbolic 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. 1 as 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-check is indispensable, as two symbolic formulas may happen to generate the same value. For example, $i << 1$ is equal to $i * i$ when $i = 2$.

4.3 Caching Equivalence Queries

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 [4] and KLEE [3]. The key of `QueryMap` is MD5 value of an equivalence query, such as whether `edx_1 = eax_3`

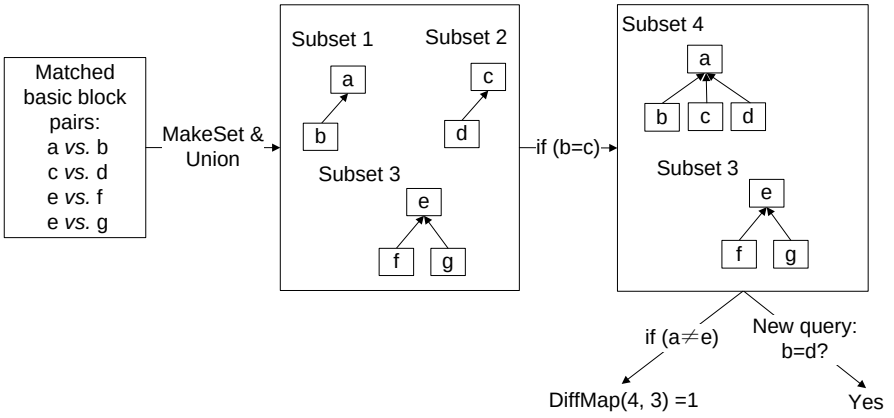

Fig. 5. Example of MakeSet-Union-Find operations

in Fig. 1; 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.

Algorithm 2 Basic Block Fast Matching

```
1: function FastMatching(a, b)                    ▷ a, b are two basic blocks to be compared
2:     a' ← Normalize(a)
3:     b' ← Normalize(b)
4:     if Hash(a') = Hash(b') then                ▷ a and b have the same instructions
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
```

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
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

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

5 Experimental Evaluation

5.1 Implementation and Experiment Setup

We have implemented our basic blocks fast matching algorithm on top of iBinHunt [15], a binary diffing tool to find semantic differences between execution traces, with about 1,800 Ocaml lines of code. The saving and loading of union-find set, DiffMap and QueryMap are implemented using the Ocaml Marshal API, which encodes arbitrary data structures as sequences of bytes and then stores them in a disk file.

We collected malware samples from VX Heavens[1] and leveraged an online malware scan service, VirusTotal[2], to classify the samples into an initial 12 families. These malware samples range from simple virus to considerably large Trojan

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

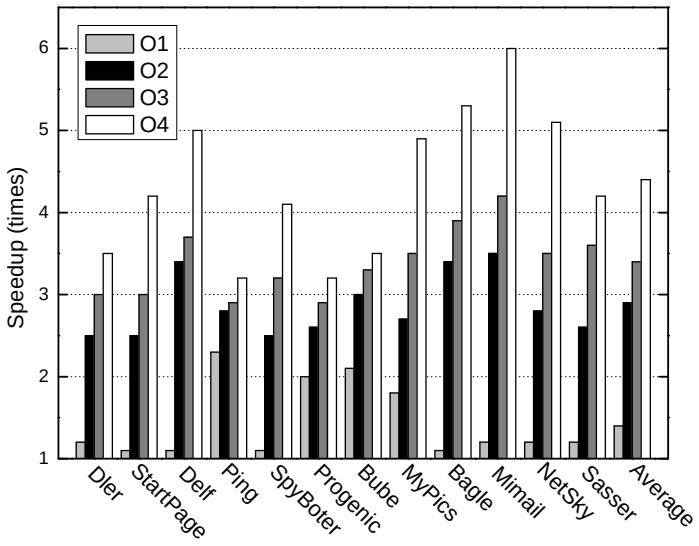

Fig. 6. The impact of basic blocks fast matching on malware lineage inference: O1 (normalization), O2 (O1 + union-find set and DiffMap), O3 (O2 + concretizing symbolic formulas), O4 (O3 + QueryMap).

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 pair-wise comparison to determine relationships among malware variants. The forth column of Table 1 lists the number of pair-wise comparison of each family and the total number is $1,008$. 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 [22], a whole-system emulator. Since most of malware are packed, we developed a generic unpacking plug-in to monitor malware sample's unpacking and start to record trace only when the execution reaches the original entry point (OEP) [13].

5.2 Performance

Cumulative Effects We first quantify the effects of the set of optimizations we presented in our basic blocks fast matching algorithm (Algorithm 2). Fig. 6 shows the speedup of malware lineage inference within each family when applying optimizations cumulatively on iBinHunt. Our baseline for this experiment is a conventional iBinHunt without any optimization we proposed. The "O1" bar indicates the effect of normalization, which can quickly identify basic block pairs with the same byte sequences. Such simple normalization only achieve notable improvement on several families such as Ping and Bube, in which instructions are quite similar in syntax. The bar denoted as "O2", captures the effect of the union-find set and DiffMap, which record previously compared results. Optimization O2 brings a significant speedup from 1.4x to 2.9x on average. Especially for some

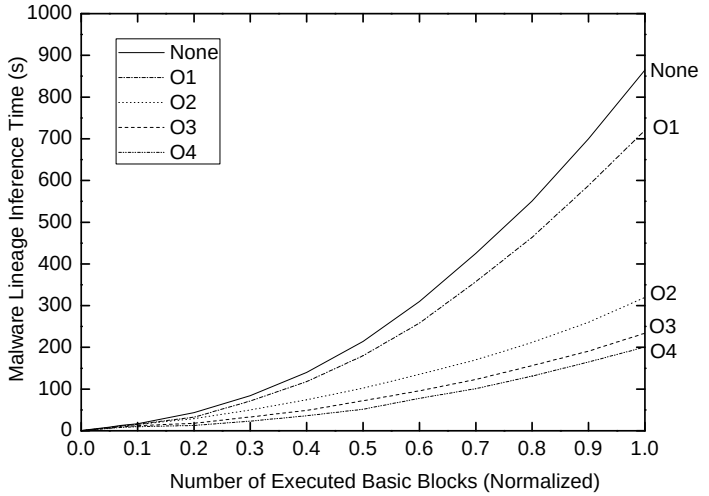

Fig. 7. The effect of our optimizations over time on Sasser family

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. Optimization of QueryMap (O4) offers an enhanced performance improvement by average 30% and with a peak value 46% to the NetSky. Particularly, since StartPage samples adopt different implementation ways to tamper with the startup page of Internet browsers, we observer quit large similarity distances among StartPage variants. In spite of this, our approach still accelerate the binary code diffing greatly.

Furthermore, we study the effect of our basic blocks fast matching over time. We choose Sasser to test because the impact of the optimizations on Sasser is close to the average value. As shown in Fig. 7, as the union-find set is enlarged and QueryMap is filled, 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.

Alleviate Performance Bottleneck In Section 3, we 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 iBinHunt. Similarly, the column 5∼7 shows the effect to reduce the number of STP invocations. In summary, our approach outperforms conventional iBinHunt in terms of less symbolic execution time by a factor of 4.0x on average, and fewer STP invocations by 4.3x on average.

Optimizations Breakdown Table 3 presents our optimizations breakdown when performing lineage inference for the four large malware families shown in

Table 2. Improvement to performance bottleneck

Malware Family	SE Times (s)			# STP Invocations		
	None	Optimization	Speedup	None	Optimization	Speedup
Dler	10.1	2.5	4.0	1123	346	3.2
StartPage	13.5	3.3	4.1	1350	314	4.3
Delf	8.8	2.0	4.4	1685	324	5.2
Ping	32.2	10.7	3.0	6740	1926	3.5
SpyBoter	16.6	4.4	3.8	2020	493	4.1
Progenic	20.2	6.3	3.2	2566	856	3.0
Bube	5.1	1.8	2.8	847	250	3.4
MyPics	11.4	2.5	4.6	1235	257	4.8
Bagle	24.4	5.1	4.8	5570	1092	5.1
Mimail	9.0	1.7	5.3	2901	484	6.0
NetSky	20.6	4.6	4.5	4958	972	5.1
Sasser	24.2	6.2	3.9	5616	1338	4.2
Average			4.0			4.3

Table 3. Optimization breakdown

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

Fig. 3. The first row shows the ratio of matched basic block pairs with the same byte sequences after normalization (line 4 in Algorithm 2). The low ratio also demonstrates the necessity of semantics-based binary diffing approach. The next two rows list statistics of union-find set, including number of union-find subsets and the maximum number of equivalent basic blocks in one subset. 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 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 more than 50% output variables comparisons. At last, we present the overall memory cost to maintain union-find set, DiffMap and QueryMap. Reassuringly, the overhead introduced by our optimizations is small.

6 Related Work

Our efforts attempt to speed up semantics-based binary diffing, which can find equivalent binary pairs that reveal syntactic differences [8, 14, 15, 17]. We have introduced the latest work in this direction in Section 2. In this section, we focus on the literature related to our optimization approach. Malware normalization

relies on ad-hoc rules to undo the obfuscations applied by malware developers [2, 5]. Our approach first performs a simple normalization to eliminate the effect of memory relocation and instruction idioms. Yang et al. [21] 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 [4] and KLEE [3], 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 enjoys a higher cache hit rate.

7 Conclusion

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 numerous malware samples. In this paper, we first studied the cruxes leading to the performance bottleneck and then proposed memoization optimization to speed up semantics-based binary diffing. The experiment on malware lineage inference demonstrated the efficacy of our optimizations with only minimal overhead. Another possible application of our approach is to study the life cycle of metamorphic malware variants. we plan to explore this direction in our future work.

Acknowledgements

This research is supported in part by the National Science Foundation (NSF) grants CCF-1320605 and CNS-1223710.

References

1. 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.
2. 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.
3. 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.
4. 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.

5. Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. Jiang Ming, Meng Pan, and Debin Gao. iBinHunt: Binary hunting with interprocedural control flow. In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)*, 2012.
16. 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.
17. Beng Heng Ng and Atul Prakash. Exposé: Discovering potential binary code reuse. In *Proceedings of the 37th IEEE Annual Computer Software and Applications Conference (COMPSAC'13)*, 2013.
18. Panda Security. Annual report 2013 summary. `http://press.pandasecurity.com/wp-content/uploads/2010/05/PandaLabs-Annual-Report_2013.pdf`, last reviewed, 12/02/2014.
19. Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys*, 46(1), 2013.
20. Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, February 2012.
21. 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.
22. 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.