BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking

Jiang Ming
The University of Texas at Arlington
jiang.ming@uta.edu

Dongpeng Xu, Yufei Jiang, and Dinghao Wu
The Pennsylvania State University
{dux103, yzj107, dwu}@ist.psu.edu

Abstract

Detecting differences between two binary executables (binary diffing), first derived from patch analysis, have been widely employed in various software security analysis tasks, such as software plagiarism detection and malware lineage inference. Especially when analyzing malware variants, pervasive code obfuscation techniques have driven recent work towards determining semantic similarity in spite of ostensible difference in syntax. Existing ways rely on either comparing runtime behaviors or modeling code snippet semantics with symbolic execution. However, neither approach delivers the expected precision. In this paper, we propose *system call sliced segment equivalence checking*, a hybrid method to identify fine-grained semantic similarities or differences between two execution traces. We perform enhanced dynamic slicing and symbolic execution to compare the logic of instructions that impact on the observable behaviors. Our approach improves existing semantics-based binary diffing by 1) inferring whether two executable binaries' behaviors are conditionally equivalent; 2) detecting the similarities or differences, whose effects spread across multiple basic blocks. We have developed a prototype, called *BinSim*, and performed empirical evaluations against sophisticated obfuscation combinations and more than 1,000 recent malware samples, including now-infamous crypto ransomware. Our experimental results show that BinSim can successfully identify fine-grained relations between obfuscated binaries, and outperform existing binary diffing tools in terms of better resilience and accuracy.

1 Introduction

An inherent challenge for reverse engineering is the source code of the program under examination is typically absent. The binary executable becomes the only available resource to be analyzed. The techniques to detect the difference between two executables (binary diffing) have been applied to a broad range of reverse engineering tasks. For example, the difference between a pre-batched binary and its updated version reveals the fixed vulnerability [23, 54], and such information can be exploited by attackers to quickly generate "1-day" exploit [9, 50]. The similarity between an intellectual property protected binary and a suspicious binary indicates a potential case of software plagiarism [41, 73]. A more appealing application emerges in malware analysis. According to the latest Panda Security Labs study [53], many malware samples in circulation are not brand new but rather evolutions of previously known malware code. Relentless malware developers typically apply various obfuscation schemes (e.g., packer, polymorphism, metamorphism, and code virtualization) [51, 57] to camouflage arresting features, circumvent malware detection, and impede reverse engineering attempts. Therefore, an obfuscation-resilient binary diffing method is of great necessity.

Pervasive code obfuscation schemes have driven binary diffing methods towards detecting semantic similarity despite syntactical difference (e.g., different instruction sequences or byte N-grams). Existing semantics-aware binary diffing can be classified into two categories. The first one compares runtime execution behaviors rather than instruction bytes. Since dynamic analysis has good resilience against code obfuscation [48], there has been a notable amount of work to measure the similarities of program behavior features, such as system call sequences and dependency graphs [6, 12, 14]. However, the program of interest may not involve unique system call sequence [73]. Furthermore, dynamic-only methods neglect subtle differences that do not reflect on the behavior change. In that case, two matched system calls may carry different meanings.

The second category relies on measuring the semantics of two pieces of binary code [54, 41, 37, 25, 43], which is usually based on basic block semantics mod-

eling. At a high level, it represents the input-output relations of a basic block as a set of symbolic formulas, which are later proved by either a constraint solver [41, 15, 25, 43], random sampling [54] or hashing [37] for equivalence. Although these tools are effective against moderate obfuscation within a basic block, such as register swapping, instruction reordering, instruction substitution, and junk code insertion [51], they exhibit a common "block-centric" limitation [13, 37]; that is, it is insufficient to capture the similarities or differences that go beyond a single basic block boundary. This issue stems from the fact that the effect of code transformations spreads across basic blocks, such as return-oriented programming encoding [40, 55], virtualization obfuscation's decode-dispatch loop [61], covert computation [59], and different implementation algorithms [56].

In this paper, we propose a hybrid method, *BinSim*, to address the limitations of existing binary diffing approaches. We attempt to identify fine-grained relations between obfuscated binary code. BinSim leverages a novel concept and technique called *System Call Sliced Segments* and their *Equivalence Checking*. This new technique relies on system or API calls[1] to slice out corresponding code segments and then check their equivalence with symbolic execution and constraint solving. Starting from the observable behavior, our approach integrates symbolic execution with dynamic backward slicing to compare the behavior-related instruction segments. We find that two matched system calls together with their arguments may carry different meanings. Our approach can answer whether two matched API calls are *conditional equivalent* [31]. Note that the behavior-related instruction segments typically bypass the boundary of a basic block so that we are more likely to detect similarities or differences that spread across basic blocks.

More precisely, we run two executables in tandem under the same input and environment to record their detailed execution data. Then, we rely on an advanced bioinformatics-inspired approach [34] to perform system call sequence alignment. After that, we trace back from the arguments of the matched system calls to determine instructions that directly (data flow) or indirectly (control flow) impact on the argument values. However, the standard dynamic slicing algorithm [80] does not suffice to operate at the obfuscated binaries. Our enhanced backward slicing considers many tricky issues and deals with obfuscation schemes that cause undesired slice explosion. Next, we calculate weakest preconditions (WP) along the dynamic slice. The resulting WP formulas accumulated in the two slices are then submitted to a constraint solver to verify whether they are equivalent. Now determining whether two matched system calls are truly equivalent under current path conditions boils down to a query of equivalence checking.

We have developed a prototype of BinSim on top of the BitBlaze [66] binary analysis platform. Experimental results on a range of advanced obfuscation schemes are encouraging. Compared with a set of existing binary diffing tools, BinSim exhibits better resilience and accuracy. We also evaluate BinSim and existing tools on more than $1,000$ recent malware samples, including highly dangerous and destructive crypto-ransomware (e.g., CryptoWall) [32, 33, 58]. The results show that BinSim can successfully identify fine-grained relations between obfuscated malware variants. We believe BinSim is an appealing method to complement existing malware defenses.

Scope and Contributions BinSim is mainly designed for fine-grained individual binary diffing analysis. It is an ideal fit for security analysts who need further investigation on two suspicious binaries. The previous work on large-scale coarse-grained malware comparison [6, 28] is orthogonal and complementary to BinSim. In summary, the contributions of this paper are as follows.

- BinSim presents a novel concept, System Call Sliced Segment Equivalence Checking, that relies on system or API calls to slice out corresponding code segments and then checks their equivalence with symbolic execution and constraint solving.
- BinSim can detect the similarities or differences across multiple basic blocks. Therefore, BinSim overcomes the "block-centric" limitation (Existing Method 2) to a great extent. Compared to dynamic-only approaches (Existing Method 1), BinSim provides more precise results, such as whether two programs' behaviors are conditionally equivalent.
- Performing dynamic slicing on the obfuscated binaries is rather tricky and complicated. The redundant instructions introduced by indirect memory access and fake control/data dependency can poison the slicing output. We improve the standard algorithm to produce more precise result.
- Unlike previous work that evaluates the efficacy of binary diffing either on different program versions [7, 25, 49], different compiler optimization levels [21, 41] or considerably moderate obfuscation [37, 41], we evaluate BinSim rigorously against sophisticated obfuscation combinations and recent malware. To the best of our knowledge, this is the first work to evaluate binary diffing in such scale.

[1]The system calls in Windows are named as native API. We also consider part of Windows API calls as a proxy for system calls.

2 Motivation and Overview

In this section, we first discuss the drawbacks of current semantics-aware binary diffing approaches. This also inspires us to propose our method. We will show C code for understanding motivating examples even though BinSim works on binary code. At last, we introduce the architecture of BinSim.

2.1 Motivation

Binary diffing methods based on behavior features (e.g., system call sequence or dependency graph) are prevalent in comparing obfuscated programs, in which the accurate static analysis is typically not feasible [48]. However, such dynamic-only approaches may disregard some real different semantics, which are usually caused by instruction level execution differences. Figure 1 presents such a counterexample, which lists three similar programs in the view of source code and their system call dependencies. Given any input $x \geq 0$, the three system call sequences (`NtCreateFile` $\rightarrow$ `NtWriteFile` $\rightarrow$ `NtClose`) together with their arguments are identical. Besides, these three system calls preserve a data flow dependency as well: one's return value is passed to another's in-argument (as shown in Figure 1(d)). Therefore, no matter comparing system call sequences or dependency graphs, these three programs reveal the same behavior. However, if we take a closer look at line 3 and 4 in Figure 1(b), the two statements are used to calculate the absolute value of x. That means the input value y for `NtWriteFile` in Figure 1(a) and Figure 1(b) differs when $x < 0$. In another word, these two programs are only *conditionally equivalent*. Note that by random testing, there is only about half chance to find Figure 1(a) and Figure 1(b) are different. Recently, the "query-then-infect" pattern has become common in malware attacks [77], which only target specific systems instead of randomly attacking victim machines. When this kind of malware happens to reveal the same behavior, dynamic-only diffing methods may neglect such subtle conditional equivalence and blindly conclude that they are equivalent under all conditions.

Another type of semantics-aware binary diffing utilizes symbolic execution to measure the semantics of the binary code. The core of current approaches is matching semantically equivalent basic blocks [25, 37, 41, 43, 54]. The property of straight-line instructions with one entry and exit point makes a basic block a good fit for symbolic execution (e.g., no path explosion). In contrast, symbolic execution on a larger scope, such as a function, has two challenges: 1) recognizing function boundary in stripped binaries [5]; 2) performance bottleneck even on the moderate size of binary code [46]. Such block-centric methods are effective in defeating instruction obfuscation within a basic block. Figure 2 presents two equivalent basic blocks whose instructions are syntactically different. Their output symbolic formulas are verified as equivalent by a constraint solver (e.g., STP [24]). However, there are many cases that the semantic equivalence spread across the boundary of a basic block. Figure 3 presents such an example, which contains three different implementations to count the number of bits in an unsigned integer (`BitCount`). Figure 3(a) and Figure 3(b) exhibit different loop bodies, while Figure 3(c) has merely one basic block. Figure 3(c) implements `BitCount` with only bitwise operators. For the main bodies of these three examples, we cannot even find matched basic blocks, but they are indeed semantically equivalent. Unfortunately, current block-centric binary diffing methods fail to match these three cases. The disassembly code of these three `BitCount` algorithms are shown in Appendix Figure 11.

Figure 4 shows another counterexample, in which the semantic difference spreads across basic blocks. When a basic block produces multiple output variables, existing block-centric binary diffing approaches try all possible permutations [54, 41, 25] to find a bijective mapping between the output variables. In this way, the two basic block pairs in Figure 4 (BB1 vs. BB1' and BB2 vs. BB2') are matched. Please note that the input variables to BB2 and BB2' are switched. If we consider the two sequential executed basic blocks as a whole, they will produce different outputs. However, the current block-centric approach does not consider the context information such as the order of matched variables. Next, we summarize possible challenges that can defeat the block-centric binary diffing methods.

1. The lack of context information such as Figure 4.

2. Compiler optimizations such as loop unrolling and function inline, which eliminate conditional branches associated.

3. Return-oriented programming (ROP) is originally designed as an attack to bypass data execution prevention mechanisms [60]. The chain of ROP gadgets will result in a set of small basic blocks. ROP has been used as an effective obfuscation method [40, 55] to clutter control flow.

4. Covert computation [59] utilizes the side effects of microprocessors to hide instruction semantics across a set of basic blocks.

5. The same algorithm but with different implementations such as Figure 3, Figure 12, and Figure 13. More examples can be found in Hacker's

```
1: int x, y; // x is an input
2: HANDLE out = CreateFile("a.txt", ...);
3: y = x + x;
4: WriteFile(out, &y, sizeof y, ...);
5: CloseHandle(out);
```

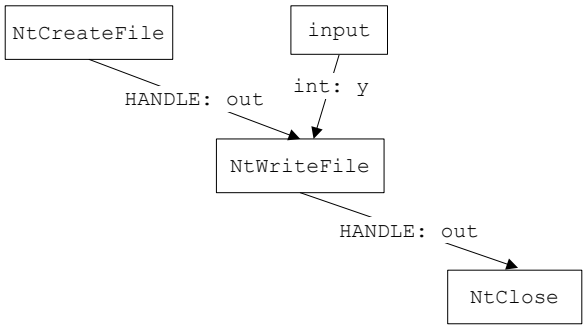
(a)

```
1: int x, y, z; // x is an input
2: HANDLE out = CreateFile("a.txt", ...);
3: z = (x >> 31);
4: z = (x ^ z) - z; // z is the absolute value of x
5  y = 2 * z;
6: WriteFile(out, &y, sizeof y, ...);
7: CloseHandle(out);
```

(b)

```
1: int x, y; // x is an input
2: HANDLE out = CreateFile ( "a.txt", ... );
3: y = x << 1;
4: WriteFile ( out, &y, sizeof y, ... );
5: CloseHandle ( out );
```

(c)



(d) System call (Windows native API) sequence and dependency

Figure 1: Example: system calls are conditional equivalent.

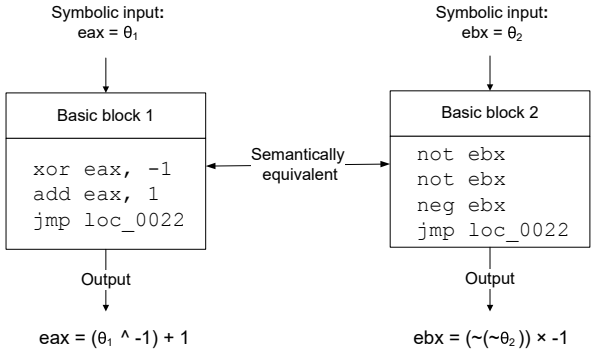


Figure 2: Semantically equivalent basic blocks with different instructions.

Delight [74], which is a collection of programming optimization tricks with bitwise operations.

6. Control flow obfuscation schemes, such as opaque predicates [17] and control flow flattening [71], can break up one basic block into multiple ones.

7. Virtualization obfuscation decode-dispatch loop [61, 79] generates a sequence of basic blocks to interpret one x86 instruction. This difficulty is further exacerbated by multi-level virtualization.

BinSim's hybrid approach can naturally break basic block boundaries and link related instructions. However, we have to take extra efforts to address the last two challenges. We will discuss them in Section 4.

2.2 Methodology

Figure 5 illustrates BinSim's core method. Given two programs P and P', our approach performs dynamic analysis as well as symbolic execution to compare how the matched system call arguments are calculated, instead of their exhibited values. We first run P and P' in tandem under the same input and environment to collect the logged traces together with their system call sequences. Then we do the system call sequences alignment to get a list of matched system call pairs (step 1). Another purpose of system call alignment is to fast identify programs exhibiting very different behaviors. After that, starting from the matched system calls arguments, we conduct backward slicing on each logged trace to identify instructions that affect the argument both directly (data flow) and indirectly (control flow). We extend the standard dynamic slicing algorithm to deal with the challenges when working on obfuscated binaries. Next, we compute the weakest precondition (WP) along each slice (step 2). In principle, WP is a symbolic formula that captures the data flow and control flow that affect the calculation of the argument. However, cryptographic functions typically generate complicated symbolic representations that could otherwise be hard to solve. To walk around this obstacle, we identify the possible cryptographic functions from the sliced segments and decom-

```
1: void BitCount1(unsigned int n)
2: {
3:   unsigned int count = 0;
4:   for (count = 0; n; n >>= 1)
5:     count += n & 1 ;
6:   printf ("%d", count);
7: }
```

(a)

```
1: void BitCount2(unsigned int n)
2: {
3:   unsigned int count = 0;
4:   while (n != 0 ) {
5:        n = n & (n-1);
6:        count++;
7:     }
8:   printf ("%d", count);
9: }
```

(b)

```
1: void BitCount3(unsigned int n)
2: {
3:   n = (n & (0x55555555)) +
         ((n >> 1) & (0x55555555));
4:   n = (n & (0x33333333)) +
         ((n >> 2) & (0x33333333));
5:   n = (n & (0x0f0f0f0f)) +
         ((n >> 4) & (0x0f0f0f0f));
6:   n = (n & (0x00ff00ff)) +
         ((n >> 8) & (0x00ff00ff));
7:   n = (n & (0x0000ffff)) +
         ((n >> 16) & (0x0000ffff));
8:   printf ("%d", n);
9: }
```

(c)

Figure 3: Semantic equivalence spreads across basic blocks.

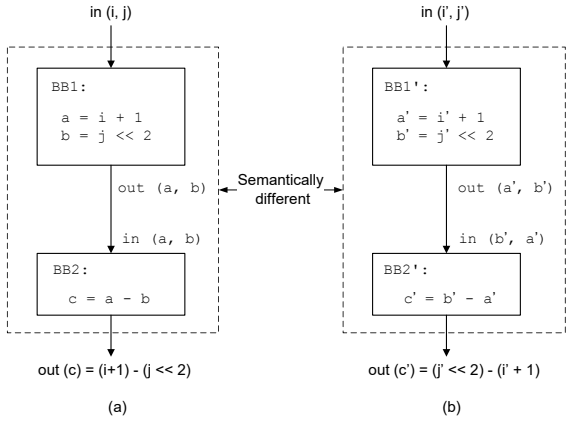


Figure 4: Semantic difference spreads across basic blocks.

pose them from equivalence checking (step 3). Then we utilize a constraint solver to verify whether two WP formulas are equivalent (step 4). Following the similar style, we compare the remaining system call pairs. At last, we perform an approximate matching on identified cryptographic functions (step 5) and calculate the finial similarity score.

Now we use the examples shown in Section 2.1 to describe how BinSim improves existing semantics-based binary diffing approaches. Assume we have got the aligned system call sequences shown in Figure 1(d). Starting from the address of the argument y in `NtWriteFile`, we do backward slicing and compute WP with respect to y. The results of the three programs are shown as follows.

$$\psi_{1a}: \; x+x$$
$$\psi_{1b}: \; 2\times((x\wedge(x>>31))-(x>>31))$$
$$\psi_{1c}: \; x<<1$$

To verify whether $\psi_{1a}=\psi_{1b}$, we check the equivalence of the following formula:

$$x+x \;=\; 2\times((x\wedge(x>>31))-(x>>31)) \quad (1)$$

Similarly, we check whether $\psi_{1a}=\psi_{1c}$ by verifying the formula:

$$x+x \;=\; x<<1 \quad (2)$$

The constraint solver will prove that Formula 2 is always true but Formula 1 is not. Apparently, we can find a counterexample (e.g., $x=-1$) to falsify Formula 1. Therefore, we have ground truth that the `NtWriteFile` in Figure 1(a) and Figure 1(c) are truly matched, while `NtWriteFile` in Figure 1(a) and Figure 1(b) are *conditionally equivalent* (when the input satisfies $x\geq 0$).

For the three different implementations shown in Figure 3, BinSim works on the execution traces under the same input (n). In this way, the loops in Figure 3 have been unrolled. Starting from the output argument, the resulting WP captures the semantics of "bits counting" across basic blocks. Therefore, we are able to verify that the three algorithms are equivalent when taking the same unsigned 32-bit integer as input. Similarly, we can verify that the two code snippets in Figure 4 are not semantically equivalent.

2.3 Architecture

Figure 6 illustrates the architecture of BinSim, which comprises two stages: online trace logging and offline comparison. The online stage, as shown in the left side of Figure 6, involves two plug-ins built on Temu [66], a whole-system emulator: *generic unpacking* and *on-demand trace logging*. Temu is also used as a malware execution sandbox in our evaluation. The recorded traces are passed to the offline stage of BinSim for comparison (right part of Figure 6). The offline stage consists of three components: preprocessing, slicing and WP calculation, and segment equivalence checker. Next, we will present each step of BinSim in the following two sections.

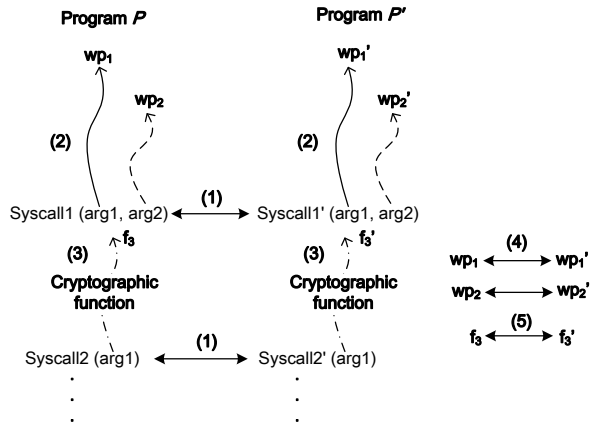


Figure 5: System call sliced segment equivalence checking steps: (1) system call alignment; (2) dynamic slicing and weakest precondition calculation; (3) cryptographic function detection; (4) equivalence checking; (5) cryptographic function approximate matching.

3 On-demand Trace Logging

BinSim's online logging stage records the needed information for the subsequent steps. The logged trace data consist of three parts: 1) instruction log contains each executed instruction's x86 opcode and values of operands; 2) memory log stores memory access addresses, which facilitate binary code slicing; 3) system calls invoked and their data flow dependencies. In general, not all of the trace data are of interest. For example, a common optimization adopted by the offline symbolic execution is "function summary" [10]. For some well-known library functions that have explicit semantics (e.g., string operation functions), we can turn off logging when executing them and generate a symbolic summary correspondingly in the offline analysis. Another example is many malware samples exhibit the malicious behavior only after the real payload is unpacked. Our generic unpacking plug-in, similar to the hidden code extractor [30], supports recording the execution trace that comes from real payload instead of various unpacking routines.

One common attack to system call recording is adding irrelevant system calls on purpose, which can also poison the system call sequences alignment. To remove system call noises, we leverage Temu's customizable multi-tag taint tracking feature to track data flow dependencies between system calls. Kolbitsch et al. [36] have observed three possible sources of a system call argument: 1) the output of a previous system call; 2) the initialized data section (e.g., `.bss` segment); 3) the immediate argument of an instruction (e.g., `push 0`). Except for the immediate argument, we label the system call outputs and the value read from the initialized data section as different taint tags. In this way, the irrelevant system calls without data dependency will be filtered out. The tainted arguments of aligned system call will be taken as the starting point of our backward slicing.

We also consider the parameter semantics. For example, although *NtClose* takes an integer as input, the source of the parameter should point to an already opened device rather than an instruction operand (see Figure 1). Therefore, the fake dependency such as "`xor eax, eax; NtClose(eax);`" will be removed. Another challenge is malware could invoke a different set of system calls to achieve the same effect. Recent work on "replacement attacks" [44] shows such threat is feasible. We will discuss possible workaround in Section 6.

4 Offline Analysis

4.1 Preprocessing

When the raw trace data arrive, BinSim first lifts x86 instructions to Vine IL. The static single assignment (SSA) style of Vine IL will facilitate tracking the use-def chain when performing backward slicing. Besides, Vine IL is also side effect free. It explicitly represents the setting of the `eflags` register bits, which favors us to identify instructions with implicit control flow and track ROP code. For example, the carry flag bit (cf) is frequently used by ROP to design conditional gadget [60].

Then we align the two collected system call sequences to locate the matched system call pairs. System call sequence alignment has been well studied in the previous literature [34, 76]. The latest work, MalGene [34], tailors Smith-Waterman local alignment algorithm [65] to the unique properties of system call sequence, such as limited alphabet and sequence branching caused by thread scheduling. Compared to the generic longest common subsequences (LCS) algorithm, MalGene delivers more accurate alignment results. There are two key scores in Smith-Waterman algorithm: *similarity function on the alphabet* and *gap penalty scheme*. MalGene customizes these two scores for better system call alignment.

Our system call alignment adopts a similar approach as MalGene [34] but extends the scope of *critical system calls*, whose alignments are more important than others. Since MalGene only considers the system call sequence deviation of the same binary under different runtime environments, the critical system calls are subject to process and thread operations. In contrast, BinSim focuses on system call sequence comparisons of polymorphic or metamorphic malware variants. Our critical system calls include more key system object operations. Appendix Table 7 lists some examples of critical system calls/Windows API we defined. Note that other system call comparison methods, such as dependency graph iso-

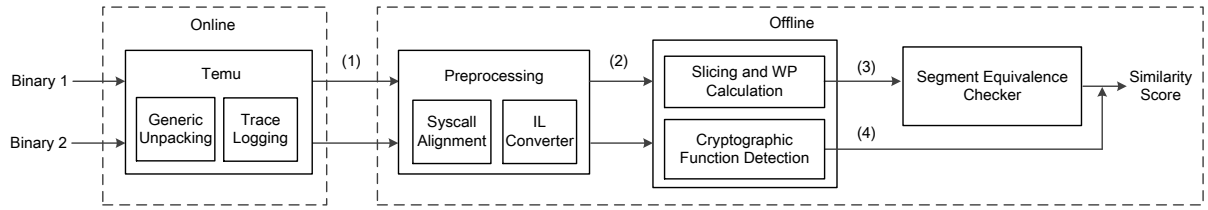


Figure 6: Schematic overview of BinSim. The output for each processing: (1) unpacked code, instruction log, memory log, and system call sequences; (2) IL traces and initial matched system call pairs; (3) weakest preconditions of system call sliced segments; (4) identified cryptographic functions.

morphism [14] and tree automata inference [3] are orthogonal to our approach.

4.2 Dynamic Slicing Binary Code

After system calls alignment, we will further examine the aligned system calls to determine whether they are truly equivalent. To this end, commencing at a tainted system call's argument, we perform dynamic slicing to backtrack a chain of instructions with data and control dependencies. The slice criterion is ⟨eip, argument⟩, while `eip` indicates the value of instruction pointer and `argument` denotes the argument taken as the beginning of backwards slicing. We terminate our backward slicing when the source of slice criterion is one of the following conditions: the output the previous system call, a constant value, or the value read from the initialized data section. Standard dynamic slicing algorithm [1, 80] relies on program dependence graph (PDG), which explicitly represents both data and control dependencies. However, compared to the source code slicing, dynamic slicing on the obfuscated binaries is never a textbook problem. The indirect memory access of binary code will pollute the conventional data flow tracking. Tracking control dependencies in the obfuscated binary code by following explicit conditional jump instructions is far from enough. Furthermore, the decode-dispatch loop of virtualization obfuscation will also introduce many fake control dependencies. As a result, conventional dynamic slicing algorithms [1, 80] will cause undesired slice explosion, which will further complicate weakest precondition calculation. Our solution is to split data dependencies and control dependencies tracking into three steps: 1) index and value based slicing that only consider data flow; 2) tracking control dependencies; 3) remove the fake control dependencies caused by virtualization obfuscation code dispatcher.

BinSim shares the similar idea as Coogan et al. [19] in that we both decouple tracing control flow from data flow when handling virtualization obfuscation. Coogan et al.'s approach is implemented through an equational reasoning system, while BinSim's dynamic slicing is built on an intermediate language (Vine IL). However, BinSim is different from Coogan et al.'s work in a number of ways, which we will discuss in Section 7.

4.2.1 Index and Value Based Slicing

We first trace the instructions with data dependencies by following the "use-def" chain (ud-chain). However, the conventional ud-chain calculation may result in the precision loss when dealing with indirect memory access, in which general registers are used to compute memory access index. There are two ways to track the ud-chain of indirect memory access, namely *index based* and *value based*. The index based slicing, like the conventional approach, follows the ud-chain related the memory index. For the example of `mov edx [4*eax+4]`, the instructions affecting the index `eax` will be added. Value based slicing, instead, considers the instructions related to the value stored in the memory slot. Therefore, the last instruction that writes to the memory location `[4*eax+4]` will be included. In most cases, the value based slicing is much more accurate. Figure 7 shows a comparison between index based slicing and value based slicing on the same trace. Figure 7(a) presents the C code of the trace. In Figure 7(b), index based slicing selects the instructions related to the computation of memory index `j = 2*i + 1`. In contrast, value based slicing in Figure 7(c) contains the instructions that is relevant to the computation of memory value `A[j] = a + b`, which is exactly the expected slicing result. However, there is an exception that we have to adopt index based slicing: when an indirect memory access is a valid index into a jump table. Jump tables typically locate at read-only data sections or code sections, and the jump table contents should not be modified by other instructions. Therefore, we switch to track the index ud-chain, like `eax` rather than the memory content.

4.2.2 Tracking Control Dependency

Next, we include the instructions that have control dependencies with the instructions in the last step. In addition to explicit conditional jump instructions (e.g., `je`

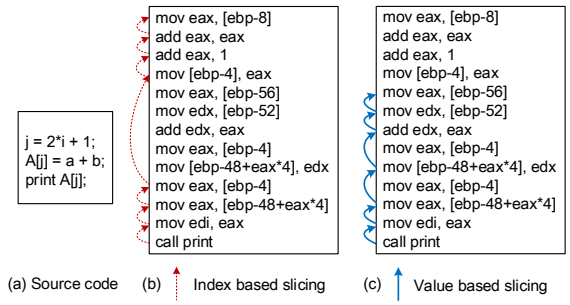


Figure 7: Index based vs. Value based slicing.

and `jne`), obfuscators may hide control flow into indirect jumps by using encoding function to calculate the real branch [78]. Our solution is to trace how the control transfer is calculated. We observe that most x86 conditional control transfers depend on certain bit value of the `eflags` register (e.g., `zf` and `cf`). Even obfuscators try to hide the conditional jumps, they still need to use arithmetic operations on certain `eflags` bits (e.g., ROP obfuscation [40, 55] and covert computation [59]). To identify these implicit control transfers, our approach trace the data flow of `eflags` bit value; that is, the instructions that calculate the bit value of the `eflags` are added into the slice. Note that in addition to the explicit conditional jump instructions, there are quite a number of instructions that have conditional jump semantics. For example, `cmovne ebx,edx` moves the value of `edx` to `ebx` according to `zf` flag. We also notice a case that the conditional logic is implemented without `eflags`: `jecxz` jumps if register `ecx` is zero. Currently BinSim supports all these special cases, which are summarized in Appendix Table 8.

4.2.3 Dispatcher Identification

Virtualization obfuscation, a great challenge to binary code slicing [61, 79], replaces the original binary code with new type of bytecode, and a specific interpreter is attached to interpret each bytecode. Due to the ease of implementing an instruction set emulator [64], current tools adopt decode-dispatch loop based interpretation [69, 52, 16]. Besides, the virtualization bytecode is designed as stack architecture style [62], which has a simple representation but requires more statements for a given computation. One instruction is typically translated to a sequence of bytecode operations on the stack values through the decode-dispatch loop. As a result, the collected slice from the above steps will contain a large number of redundant instructions caused by decode-dispatch loop iterations. We observe that each decode-dispatch loop iteration has the following common features.

1. It is a sequence of memory operations, ending with an indirect jump.
2. It has an input register `a` as virtual program counter (VPC) to fetch the next bytecode (e.g., `ptr[a]`). For example, VMProtect [69] takes `esi` as VPC while Code Virtualizer [52] chooses `al` register.
3. It ends with an indirect jump which dispatches to a bytecode handler table. The index into the jump table has a data dependency with the value of `ptr[a]`.

Our containment technique is to first identify possible decode-dispatch loop iterations in the backward slice according to the above common features. For each instruction sequence ending with an indirect jump, we mark the input registers as $a_1, a_2, ...a_n$ and output registers as $b_1, b_2, ...b_n$. Then we check whether there is an output register b_i meets the two heuristics:

1. b_i is tainted by the data located in `ptr[`a_j`]`.
2. The instruction sequence ends with `jmp ptr [`b_i`*(table stride) + table base]`.

After that, we will remove the fake control dependencies caused by virtualization obfuscation code dispatcher. In our preliminary testing, five virtualization obfuscation protected instructions produces as many as 3,163 instructions, and most of them are related to the decode-dispatch loop. After our processing, the number of instruction is reduced to only 109.

4.3 Handling Cryptographic Functions

Now-infamous crypto ransomware extort large ransom by encrypting the infected computer's files with standard cryptographic functions [33]. One ransomware archetype typically evolves from generation to generation to produce a large number of new strains. They may refine old versions incrementally to better support new criminal business models. In addition to the generic detection methods based on monitoring file system anomalies [32, 58], it is very interesting to investigate this emerging threat with BinSim, such as identifying ransomware variant relationships and investigate ransomware evolution. However, cryptographic functions have been known to be a barrier to SMT-based security analysis in general [11, 72] because of the complicated input-output dependencies. Our backward slicing step will produce a quite long instruction segment, and the corresponding equivalence checking will become hard to solve as well [67].

We observer that cryptographic function execution has almost no interaction with system calls except the ones are used for input and output. For example, crypto ransomware take the original user's file as input, and then

overwrite it with the encrypted version. Inspired by Caballero et al.'s work [11], we do a "stitched symbolic execution" to walk around this challenge. Specifically, we first make a forward pass over the sliced segments to identify the possible cryptographic functions between two system calls. We apply the advanced detection heuristics proposed by Gröbert et al. [27] (e.g., excessive use of bitwise operations, instruction chains, and mnemonic const values) to quickly match known cryptographic function features. If a known cryptographic function is detected, we will turn off the weakest precondition calculation and equivalence checking. In Section 4.5, we will discuss how to approximately measure the similarity of detected cryptographic functions.

4.4 Weakest Precondition Calculation

Let's assume the slice we collected (S) contains a sequence of instructions $[i_1, i_2, ..., i_n]$. Our weakest precondition (WP) calculation takes (S) as input, and the state of the execution to the given API call's argument as the postcondition (P). Inductively, we first calculate $\text{wp}(i_n, P) = P_{n-1}$, then $\text{wp}(i_{n-1}, P_{n-1}) = P_{n-2}$ and until $\text{wp}(i_1, P_1) = P_0$. The weakest precondition, denoted as $\text{wp}(S, P) = P_0$, is a boolean formula over the inputs that follow the same slice (S) and forces the execution to reach the given point satisfying P. We adopt a similar algorithm as Banerjee et al.'s [4] to compute the WP for every statement in the slice, following both data dependency and control dependency. The resultant WP formula for a program point can be viewed as a conjunction of predicates accumulated before that point, in the following form:

$$WP = F_1 \wedge F_2 \wedge ... \wedge F_k.$$

Opaque predicates [17], a popular control flow obfuscation scheme, can lead to a very complicated WP formula by adding infeasible branches. We apply recent opaque predicate detection method [45] to identify so called invariant, contextual, and dynamic opaque predicates. We remove the identified opaque predicate to reduce the size of the WP formula.

4.5 Segment Equivalence Checking

We identify whether two API calls are semantically equivalent by checking the equivalence of their arguments' weakest preconditions. To this end, we perform *validity checking* for the following formula.

$$wp_1 \equiv wp_2 \wedge arg_1 = arg_2 \tag{3}$$

Different from existing block-centric methods, whose equivalence checking is limited at a single basic block level, our WP calculation captures the logic of a segment of instructions that go across the boundaries of basic blocks. Our method can offer a logical explanation of whether syntactically different instruction segments contribute to the same observable behavior. Frequent invocation of constraint solver imposes a significant overhead. Therefore, we maintain a HashMap structure to cache the results of the previous comparisons for better performance.

To quantitatively represent different levels of similarity and facilitate our comparative evaluation, we assign different scores ($0.5 \sim 1.0$) based on the already aligned system call sequences. The similarity sore is tuned with our ground truth dataset (Section 5.2) by two metrics: *precision* and *recall*. The precision is to measure how well BinSim identifies different malware samples; while recall indicates how well BinSim recognizes the same malware samples but with various obfuscation schemes. An optimal similarity sore should provide high precision and recall at the same time. We summarize the selection of similarity score as follows.

1. 1.0: the arguments of two aligned system calls pass the equivalence checking. Since we have confidence these system calls should be perfectly matched, we represent their similarity with the highest score.

2. 0.7: the sliced segments of two aligned system calls are corresponding to the same cryptographic algorithm (e.g. AES vs. AES). We assign a slightly lower score to represent our approximate matching of cryptographic functions.

3. 0.5: the aligned system call pairs do not satisfy the above conditions. The score indicates their arguments are either conditionally equivalent or semantically different.

Assume the system call sequences collected from program a and b are T_a and T_b, and the number of aligned system calls is n. We define the similarity calculation as follows.

$$\text{Sim}(a,b) = \frac{\sum_{i=1}^{n} Similarity\ Score}{\text{Avg}\{|T_a|, |T_b|\}} \tag{4}$$

$\sum_{i=1}^{n} Similarity\ Score$ sums the similarity score of aligned system call pairs. To balance the different length of T_a and T_b and be sensitive to system call noises insertion, we use the average number of two system call sequences as the denominator. The value of $\text{Sim}(a,b)$ ranges from 0.0 to 1.0. The higher $\text{Sim}(a,b)$ value indicates two traces are more similar.

Table 1: Different obfuscation types and their examples.

	Type	Examples
1	Intra-basic-block	Register swapping, junk code, instructions substitution and reorder
2	Control flow	Loop unrolling, opaque predicates, control flow flatten, function inline
3	ROP	Synthetic benchmarks collected from the reference [79]
4	Different implementations	BitCount (Figure 3) isPowerOfTwo (Appendix Figure 12) flp2 (Appendix Figure 13)
5	Covert computation[59]	Synthetic benchmarks
6	Single-level virtualization	VMProtect [69]
7	Multi-level virtualization	Synthetic benchmarks collected from the reference [79]

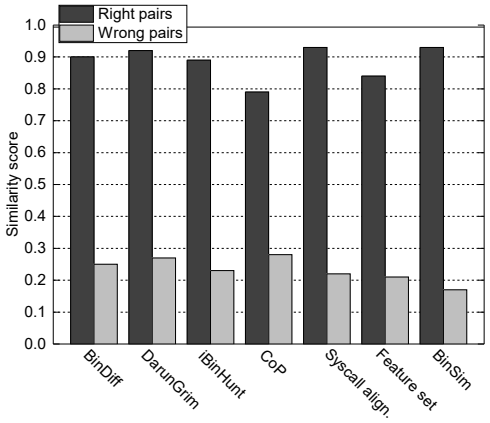


Figure 8: Similarity scores change from right pairs to wrong pairs.

5 Experimental Evaluation

We conduct our experiments with several objectives. First and foremost, we want to evaluate whether BinSim outperforms existing binary diffing tools in terms of better obfuscation resilience and accuracy. To accurately assess comparison results, we design a controlled dataset so that we have a ground truth. We also study the effectiveness of BinSim in analyzing a large set of malware variants with intra-family comparisons. Finally, performance data are reported.

5.1 Experiment Setup

Our first testbed consists of Intel Core i7-3770 processor (Quad Core with 3.40GHz) and 8GB memory, running Ubuntu 14.04. We integrate *FakeNet* [63] into Temu to simulate the real network connections, including DNS, HTTP, SSL, Email, FTP etc. We carry out the large-scale comparisons for malware variants in the second testbed, which is a private cloud containing six instances running simultaneously. Each instance is equipped with a duo core, 4GB memory, and 20GB disk space. The OS and network configurations are similar to the first testbed. Before running a malware sample, we reset Temu to a clean snapshot to eliminate the legacy effect caused by previous execution (e.g., modify registry configuration). To limit the possible time-related execution deviations, we utilize Windows Task Scheduler to run each test case at the same time.

5.2 Ground Truth Dataset

Table 1 lists obfuscation types that we plan to evaluate and their examples. Intra-basic-block obfuscation methods (Type 1) have been well handled by semantics-based binary diffing tools. In Section 2.1, we summarize possible challenges that can defeat the block-centric binary diffing methods, and Type 2 $\sim$ Type 7 are corresponding to such examples. We collect eight malware source code with different functionalities from VX Heavens[2]. We investigate the source code to make sure they are different, and each sample can fully exhibit its malicious behavior in the runtime. Besides, we also collect synthetic benchmarks from the previous work. The purpose is to evaluate some obfuscation effects that are hard to automate. Our controlled dataset statistics are shown in Table 2. The second column of Table 2 lists different obfuscation schemes and combinations we applied.

In addition to BinSim, we also test other six representative binary diffing tools. BinDiff [23] and DarunGrim [50] are two popular binary diffing products in industry. They rely on control flow graph and heuristics to measure similarities. CoP [41] and iBinHunt [43] represent "block-centric" approaches. Based on semantically equivalent basic blocks, iBinHunt compares two execution traces while CoP identifies longest common subsequence with static analysis. System call alignment and feature set are examples of dynamic-only approaches. "Feature set" indicates the method proposed by Bayer et al. [6] in their malware clustering work. They abstract system call sequence to a set of features (e.g., OS object, OS operations, and dependencies) and measure the similarities of two feature sets by Jaccard Index. For comparison, we have implemented the approaches of CoP [41], iBinHunt [43], and feature set [6]. The system call alignment is the same to the method adopted by BinSim.

5.3 Comparative Evaluation Results

Naively comparing these seven binary diffing tools with their similarity scores is not informative[3]. It is also very difficult to interpret precision and recall values because each tool adopts different similarity metrics and thresh-

[2] http://vxheaven.org/src.php

[3] We have normalized all the similarity scores from 0.0 $\sim$ 1.0.

Table 2: Controlled dataset statistics. The obfuscation type numbers are defined in Table 1.

Sample	Obfuscation type	LoC #	Online (Normalized)	Offline (min)		
				Preprocess	Slicing & WP	STP (no/opt)
Malware						
BullMoose	6	30	5X	1	2	1/0.5
Clibo	1+6	90	6X	1	2	2/0.8
Branko	1+2+6	270	8X	2	3	3/1
Hunatcha	2	340	8X	2	4	1/1
WormLabs	1	420	8X	2	6	3/2
KeyLogger	2	460	12X	2	6	4/2
Sasser	1+2+6	950	9X	3	8	4/3
Mydoom	1+2	3276	10X	3	10	6/4
Synthetic benchmark						
ROP	3	449	6X	1	3	2/1
Different implementations	4	80	6X	1	2	2/0.8
Covert computation	5	134	6X	1	2	3/1
Multi-level virtualization	7	140	10X	4	12	5/3

Table 3: Absolute difference values of similarity scores under different obfuscation schemes and combinations.

Sample	Obfuscation type	BinDiff	DarunGrim	iBinHunt	CoP	Syscall alignment	Feature set	BinSim
BullMoose	6	0.58	0.56	0.39	0.61	0.08	0.10	0.08
Clibo	1+6	0.57	0.64	0.41	0.62	0.10	0.12	0.10
Branko	1+2+6	0.63	0.62	0.35	0.68	0.10	0.15	0.12
Hunatcha	2	0.40	0.42	0.19	0.30	0.12	0.17	0.12
WormLabs	1	0.10	0.12	0.03	0.03	0.08	0.12	0.05
KeyLogger	2	0.38	0.39	0.12	0.26	0.09	0.15	0.09
Sasser	1+2+6	0.62	0.62	0.42	0.58	0.12	0.18	0.10
Mydoom	1+2	0.42	0.38	0.10	0.38	0.10	0.15	0.05
ROP	3	0.63	0.54	0.49	0.52	0.10	0.10	0.10
Different implementations	4	0.48	0.39	0.48	0.52	0.05	0.10	0.05
Covert computation	5	0.45	0.36	0.44	0.45	0.05	0.10	0.05
Multi-level virtualization	7	0.68	0.71	0.59	0.69	0.15	0.18	0.16
Average								
"Right pairs" vs. "Obfuscation pairs"		0.50	0.48	0.34	0.46	0.10	0.15	0.09
"Right pairs" vs. "Wrong pairs"		0.65	0.65	0.66	0.55	0.71	0.63	0.76

old. What matters is that a tool can differentiate right-pair scores from wrong-pair scores. We first test how their similar scores change from right pairs to wrong pairs. For the right pair testing, we compare each sample in Table 2 with itself (no obfuscation). The average values are shown in "Right pairs" bar in Figure 8. Then we compare each sample with the other samples (no obfuscation) and calculate the average values, which are shown in "Wrong pairs" bar in Figure 8[4]. The comparison results reveal a similar pattern for all these seven binary diffing tools: a large absolute difference value between the right pair score and the wrong pair score.

Next, we figure out how the similarity score varies under different obfuscation schemes and combinations. We first calculate the similarity scores for "Right pairs" (self comparison) and "Obfuscation pairs" (the clean version vs. its obfuscated version). Table 3 shows the absolute difference values between "Right pairs" and "Obfuscation pairs". Since code obfuscation has to preserve semantics [17], the small and consistent difference values can indicate that a binary diffing tool is resilient to different obfuscation schemes and combinations. BinDiff, DarunGrim, iBinHunt and CoP do not achieve a consistent (good) result for all test cases, because their difference values fluctuate. The heuristics-based comparisons adopted by BinDiff and DarunGrim can only handle mild instructions obfuscation within a basic block. Since multiple obfuscation methods obfuscate the structure of control flow graph (e.g., ROP and control flow obfuscation), the effect of BinDiff and DarunGrim are limited. CoP and iBinHunt use symbolic execution and theorem proving techniques to match basic blocks, and therefore are resilient to intra-basic-block obfuscation (Type 1). However, they are insufficient to defeat the obfuscation that may break the boundaries of basic blocks (e.g., Type 2 $\sim$ Type 7 in Table 1). The last rows of Table 3 shows the average difference values for the "Right pairs" vs. "Obfuscation pairs" and "Right pairs" vs. "Wrong pairs". The closer for these two scores, the harder for a tool to set a threshold or cutoff line to give a meaningful information on the similarity score.

[4]It does not mean that higher is better on the similarity scores for the right pairs, and lower is better for the wrong pairs. What is important is how their similarity values change from right pairs to wrong pairs.

Table 4: Comparison of slice sizes (# instructions).

Sample	Obfuscation type	No-VMP	Conventional	BinSim
BullMoose	6	98	6,785	165
Clibo	1+6	156	16,860	238
Branko	1+2+6	472	31,154	520
Sasser	1+2+6	1,484	64,276	1,766
fibonacci	7	156	4,142	278

Table 5: Similarity score of four CryptoWall variants.

a vs. b	a vs. c	a vs. d	b vs. c	b vs. d	c vs. d
0.92	0.83	0.32	0.78	0.33	0.37

Regarding dynamic-only methods (system call alignment and feature set), their scores are consistent for most comparisons. The reason is dynamic-only approaches are effective to defeat most code obfuscation schemes. However, we notice a variant of `Hunatcha` worm exhibits the malicious behavior under the condition of *systime.Month* < 12. Without more detailed information such as path conditions, both system call alignment and feature set methods fail to identify such conditional equivalence. This disadvantage is further manifested by our large-scale malware comparisons, in which we find out 11% variants are conditionally equivalent.

5.4 Offline Analysis Evaluation

In this section, we first evaluate BinSim's dynamic slicing when handling obfuscated binaries. We test BinSim with VMProtect [69], an advanced commercial obfuscator. In addition to virtualization obfuscation, which can cause slice size explosion, VMProtect also performs intra-basic-block (Type 1) and control flow obfuscation (Type 2). As shown in Table 4, we obfuscate the test cases with multiple obfuscation combinations and multi-level virtualization (Type 7). "No-VMP" column indicates BinSim's result without obfuscation. The last two columns show the slice sizes of conventional dynamic slicing and BinSim. BinSim outperforms the conventional approach by reducing slice sizes significantly. Note that the sliced segment produced by BinSim contains many different instructions with "No-VMP" version. Directly comparing the syntax of instructions is not feasible. Our semantics-based equivalence checking can show that the new sliced segment is equivalent to the original instructions.

Next, we evaluate BinSim's cryptographic function approximate matching, which allows equivalence checking in the presence of cryptographic functions that could otherwise be hard to analyze. We collect four CryptoWall variants and apply BinSim to compare them pair by pair. CryptoWall is a representative ransomware family, and it is also continuously evolving. The similar scores are shown in Table 5. We notice three samples (a, b, and c) are quite similar, and one sample (CryptoWall.d) has relatively large differences with the others. After investigating BinSim's output, we find out that CryptoWall.d reveals three distinct behaviors: 1) "query-then-infect": it will terminate execution if the infected machine's UI languages are Russian, Ukrainian or other former Soviet Union country languages (via `GetSystemDefaultUILanguage`). This clearly shows that the adversaries want to exclude certain areas from attacking. 2) It uses AES for file encryption while the other three variants choose RSA. 3) It encrypts files with a new file name generation algorithm. Our "query-then-infect" findings coincide with the recent CryptoWall reverse engineering report [2].

5.5 Analyzing Wild Malware Variants

We report our experience of applying BinSim and other six binary diffing tools on $1,050$ active malware samples (uncontrolled dataset)[5]. The dataset is retrieved from VirusShare[6] and analyzed at February 2017. We leverage VirusTotal[7] to do an initial clustering by majority voting. The total $1,050$ samples are grouped into 112 families, and more than 80% samples are protected by different packers or virtualization obfuscation tools. For each binary diffing tool, we perform intra-family pairwise comparison on our private cloud. The distribution of similarity scores is shown in Table 6. Because BinDiff, DarunGrim, and CoP cannot directly work on the packed binary, we provide the unpacker binaries preprocessed by BinSim's generic unpacking.

In most cases, dynamic-only methods and BinSim are able to find small distances among intra-family samples. For example, over 86% of the pairs have a similarity score of 0.6 or greater. System call alignment has a better distribution than BinSim during the similarity score range $0.70 \sim 1.00$. We attribute the high score to the fact that system call alignment cannot detect *conditional equivalence*. Actually, we successfully identify that about 11% of malware samples have so-called "query-then-infect" behaviors [77], and BinSim is able to find whether two malware variants are conditionally equivalent. In these cases, BinSim's lower scores better fit the ground truth. Figure 9 shows a conditional equivalent behavior we find in `Trojan-Spy.Win32.Zbot` variants. Figure 10 presents a common compiler optimization that converts a high-level branch condition into a purely arithmetic sequence. This optimization can frustrate "block-centric" binary diffing methods, and we have

[5]The initial dataset is much larger, but we only consider the active samples that we can collect system calls.

[6]http://virusshare.com/

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

```
...
// modify registry key
1: RegOpenKeyEx(...);
2: RegSetValueEx(...);
3: RegCloseKey (...);
```

(a) Zbot.a

```
...
// modify registry key
1: GetLocalTime(&systime);
2: if ( systime.Day < 20)
3:  {
4:      RegOpenKeyEx(...);
5:      RegSetValueEx(...);
6:      RegCloseKey (...);
7:  }
```

(b) Zbot.b

Figure 9: Conditional equivalent behaviors between Trojan-Spy.Win32.Zbot variants.

```
if (reg)
   reg = val1;
else
   reg = val2;
```

(a) Branch logic

```
1: neg     reg
2: sbb     reg, reg
3: and     reg, (val1 - val2)
4: add     reg, val2
```

(b) Equivalent branchless logic

Figure 10: Example: branchless logic code (`reg` stands for a register; `val1` and `val2` are two inputs).

seen such cases repeatedly in our dataset. By contrast, BinSim's hybrid approach naturally identifies the implicit control dependency in Figure 10 (b).

5.6 Performance

In Table 2, we also report the performance of BinSim when analyzing the controlled dataset. The fourth column lists the runtime overhead imposed by our online trace logging. On average, it incurs 8X slowdown, with a peak value 12X when executing KeyLogger. The fifth to seventh columns present the execution time of each component in our offline analysis stage. The number of instructions in the system call slice ranges from 5 to 138 and the average number is 22. The "STP" column presents average time spent on querying STP when comparing two programs. Here we show the time before and after the optimization of caching equivalence queries. On average, the HashMap speeds up STP processing time by a factor of 1.7. Considering that BinSim attempts to automatically detect obfuscated binary code similarity, which usually takes exhausting manual efforts from several hours to days, this degree of slowdown is acceptable. Performing the intra-family comparisons on $1,050$ malware samples required approximately 3 CPU days.

6 Discussion

Like other malware dynamic analysis approaches, BinSim bears with the similar limitations: 1) incomplete path coverage; 2) environment-sensitive malware [34, 35] which can detect sandbox environment. Therefore, BinSim only detects the similarities/differences exhibiting during execution. The possible solutions are to explore more paths by automatic input generation [26, 47] and analyze malware in a transparent platform (e.g., VMRay Analyzer [70]). Our current generic unpacking is sufficient for our experiments. However, it can be defeated by more advanced packing methods such as multiple unpacking frames and parallel unpacking [68]. We plan to extend BinSim to deal with the advanced packing methods. Recent work proposes "replacement attacks" [44] to mutate system calls and their dependencies. As a result, similar malware variants turn out to have different behavior-based signatures. We regard this "replacement attacks" as a potential threat because it can reduce BinSim's similarity score. One possible solution is to design a layered architecture to capture alternative events that achieve the same high-level functionality.

BinSim's enhanced slicing algorithm handles the obfuscations that could break the block-centric binary comparisons. We have evaluated BinSim against a set of sophisticated commercial obfuscation tools and advanced academic obfuscation methods. However, determined adversaries may carefully add plenty of redundant dependencies to cause slice size explosion, and the resulting weakest preconditions could become too complicated to be solved. As an extreme case, the dependencies of a system call argument can be propagated to the entire program. To achieve this, it requires that future attackers have much deeper understanding about program analysis (e.g., inter-procedure data/control follow analysis) and take great engineering efforts. An attacker can also customize an unknown cryptographic algorithm to evade our cryptographic function approximate matching. However, correctly implementing a cryptographic algorithm is not a trivial task, and most cryptographic functions are reused from open cryptographic libraries, such as OpenSSL and Microsoft Cryptography API [75]. Therefore, BinSim raises the attacking bar significantly compared to existing techniques. On the other side, designing a worst case evaluation metric needs considerable insights into malicious software industry [39]. We leave it as our future work.

7 Related Work

7.1 Dynamic Slicing and Weakest Precondition Calculation

As dynamic slicing techniques [1, 80] can substantially reduce the massive program statements under investigation to a most relevant subset, they have been widely applied to the domain of program analysis and verification. Differential Slicing [29] produces a causal difference graph that captures the input differences leading to

Table 6: Similarity score distribution (%) of intra-family comparisons.

Score range	BinDiff	DarunGrim	iBinHunt	CoP	Syscall alignment	Feature set	BinSim
0.00–0.10	1	1	1	1	1	1	1
0.10–0.20	3	2	1	3	1	1	1
0.20–0.30	3	4	2	4	1	2	1
0.30–0.40	13	14	3	10	1	3	1
0.40–0.50	18	17	5	18	2	3	2
0.50–0.60	14	16	18	13	3	4	4
0.60–0.70	17	13	17	14	6	16	11
0.70–0.80	13	15	20	16	26	27	24
0.80–0.90	9	10	15	11	21	16	19
0.90–1.00	9	8	18	10	38	27	36

the execution differences. Weakest precondition (WP) calculation is firstly explored by Dijkstra [20] for formal program verification. Brumley et al. [8] compare WP to identify deviations in different binary implementations for the purpose of error detection and fingerprint generation. Ansuman et al. [4] calculate WP along the dynamic slicing to diagnose the root of an observable program error. BinSim's dynamic slicing and WP calculation are inspired by Ansuman et al.'s work. However, we customize our dynamic slicing algorithm to operate at the obfuscated binaries, which is more tricky than working on source code or benign programs. Another difference is we perform equivalence checking for WP while they do implication checking.

The most related backward slicing method to BinSim is Coogan et al.'s work [19]. We both attempt to identify the relevant instructions that affect system call arguments in an obfuscated execution trace, and the idea of value based slicing and tracking control dependency is similar. However, BinSim is different from Coogan et al.'s work in a number of ways. First, Coogan et al.'s approach is designed only for virtualization obfuscation. To evaluate the accuracy of backward slicing, they compare the x86 instruction slicing pairs by the syntax of the opcode (e.g., *mov*, *add*, and *lea*). It is quite easy to generate a syntactically different trace through instruction-level obfuscation [51]. Furthermore, the commercial virtualization obfuscators [52, 69] have already integrated code mutation functionality. Therefore, Coogan et al.'s approach has less resilience to other obfuscation methods. Second, we utilize taint analysis to identify virtualization bytecode dispatcher while Coogan et al. apply different heuristics. Third, Coogan et al. do not handle cryptographic functions. They state that the encryption/decryption routine could cripple their analysis. Fourth, Coogan et al. evaluate their method on only six tiny programs; while BinSim goes through an extensive evaluation. Last, but not the least, after the sub-traces or sliced segments are constructed, Coogan et al. compare them syntactically while BinSim uses weakest precondition to compare them semantically.

7.2 Binary Diffing

Hunting binary code difference have been widely applied in software security. BinDiff [23] and DarunGrim [50] compare two functions via the maximal control flow subgraph isomorphism and match the similar basic blocks with heuristics. BinSlayer [7] improves BinDiff by matching bipartite graphs. dicovRE [22] extracts a set of syntactical features to speed up control flow subgraph isomorphism. These approaches gear toward fast matching similar binary patches, but they are brittle to defeat the sophisticated obfuscation methods. Another line of work captures semantic equivalence between executables. BinHunt [25] first leverages symbolic execution and theorem proving to match the basic blocks with the same semantics. BinJuice [37] extracts the semantic abstraction for basic blocks. Exposé [49] combines function-level syntactic heuristics with semantics detection. iBinHunt [43] is an inter-procedural path diffing tool and relies on multi-tag taint analysis to reduce possible basic block matches. Pewny et al. [54] adopt basic block semantic representation sampling to search cross-architecture bugs. As we have demonstrated, these tools suffer from the so called "block-centric" limitation. In contrast, BinSim can find equivalent instruction sequences across the basic block boundary. Egele et al. [21] proposed blanket execution to match similar functions in binaries using dynamic testing. However, blanket execution requires a precise function scope identification, which is not always feasible for obfuscated binary code [42].

7.3 Malware Dynamic Analysis

Malware dynamic analysis techniques are characterized by analyzing the effects that the program brings to the operating system. Compared with static analysis, dynamic analysis is less vulnerable to various code obfuscation methods [48]. Christodorescu et al. [14] proposed to use data-flow dependencies among system calls as malware specifications, which are hard to be circum-

vented by random system calls injection. Since then, there has been a significant amount of work on dynamic malware analysis, e.g., malware clustering [6, 28] and detection [3, 12]. However, dynamic-only approaches may disregard the conditional equivalence or the subtle differences that do not affect system call arguments. Therefore, BinSim's hybrid approach is much more accurate. In addition, dynamic slicing is also actively employed by various malware analysis tasks. The notable examples include an efficient malware behavior-based detection that executes the extracted slice to match malicious behavior [36], extracting kernel malware behavior [38], generating vaccines for malware immunization [76], and identifying malware dormant functionality [18]. However, all these malware analysis tasks adopt the standard dynamic slicing algorithms [1, 80], which are not designed for tracking the data and control dependencies in a highly obfuscated binary, e.g., virtualization-obfuscated malware. As we have demonstrated in Section 4.2, performing dynamic slicing on an obfuscated binary is challenging. Therefore, our method is beneficial and complementary to existing malware defense.

8 Conclusion

We present a hybrid method combining dynamic analysis and symbolic execution to compare two binary execution traces for the purpose of detecting their fine-grained relations. We propose a new concept called *System Call Sliced Segments* and rely on their *Equivalence Checking* to detect fine-grained semantics similarity. By integrating system call alignment, enhanced dynamic slicing, symbolic execution, and theorem proving, our method compares the semantics of instruction segments that impact on the observable behaviors. Compared to existing semantics-based binary diffing methods, our approach can capture the similarities, or differences, across basic blocks and infer whether two programs' behaviors are conditionally equivalent. Our comparative evaluation demonstrates BinSim is a compelling complement to software security analysis tasks.

9 Acknowledgments

We thank the Usenix Security anonymous reviewers and Michael Bailey for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) under grants CCF-1320605 and CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2265 and N00014-16-1-2912. Jiang Ming was also supported by the University of Texas System STARs Program.

References

[1] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. *ACM SIGPLAN Notices 25*, 6 (1990), 246–256.

[2] ALLIEVI, A., UNTERBRINK, H., AND MERCER, W. CryptoWall 4 - the evolution continues. Cisco White Paper, 2016 May.

[3] BABIĆ, D., REYNAUD, D., AND SONG, D. Malware analysis with tree automata inference. In *Proceedings of the 23rd Int. Conference on Computer Aided Verification (CAV'11)* (2011).

[4] BANERJEE, A., ROYCHOUDHURY, A., HARLIE, J. A., AND LIANG, Z. Golden implementation driven software debugging. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)* (2010).

[5] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. ByteWeight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (2014).

[6] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'09)* (2009).

[7] BOURQUIN, M., KING, A., AND ROBBINS, E. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW '13)* (2013).

[8] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th USENIX Security Symposium* (2007).

[9] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposimu on Security and Privacy (S&P'08)* (2008).

[10] CABALLERO, J., MCCAMANT, S., BARTH, A., AND SONG, D. Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Tech. rep., EECS Department, University of California, Berkeley, March 2009.

[11] CABALLERO, J., POOSANKAM, P., MCCAMANT, S., BABI Ć, D., AND SONG, D. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)* (2010).

[12] CANALI, D., LANZI, A., BALZAROTTI, D., KRUEGEL, C., CHRISTODORESCU, M., AND KIRDA, E. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)* (2012).

[13] CHANDRAMOHAN, M., XUE, Y., XU, Z., LIU, Y., CHO, C. Y., AND KUAN, T. H. B. BinGo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'16)* (2016).

[14] CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. Mining specifications of malicious behavior. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* (2007).

[15] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)* (2005).

[16] COLLBERG, C. The tigress c diversifier/obfuscator. `http://tigress.cs.arizona.edu/`, last reviewed, 02/16/2017.

[17] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. Tech. rep., The University of Auckland, 1997.

[18] COMPARETTI, P. M., SALVANESCHI, G., KIRDA, E., KOLBITSCH, C., KRUEGEL, C., AND ZANERO, S. Identifying dormant functionality in malware programs. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P'10)* (2010).

[19] COOGAN, K., LU, G., AND DEBRAY, S. Deobfuscation of virtualization-obfuscated software. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)* (2011).

[20] DIJKSTRA, E. W. *A Discipline of Programming*, 1st ed. Prentice Hall PTR, 1997.

[21] EGELE, M., WOO, M., CHAPMAN, P., AND BRUMLEY, D. Blanket Execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security'14)* (2014).

[22] ESCHWEILER, S., YAKDAN, K., AND GERHARDS-PADILLA, E. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS'16)* (2016).

[23] FLAKE, H. Structural comparison of executable objects. In *Proceedings of the 2004 GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'04)* (2004).

[24] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *Proceedings of the 2007 International Conference in Computer Aided Verification (CAV'07)* (2007).

[25] GAO, D., REITER, M. K., AND SONG, D. BinHunt: Automatically finding semantic differences in binary programs. In *Poceedings of the 10th International Conference on Information and Communications Security (ICICS'08)* (2008).

[26] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (2008).

[27] GRÖBERT, F., WILLEMS, C., AND HOLZ, T. Automated identification of cryptographic primitives in binary programs. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11)* (2011).

[28] JANG, J., BRUMLEY, D., AND VENKATARAMAN, S. BitShred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS'11)* (2011).

[29] JOHNSON, N. M., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential Slicing: Identifying causal execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P'11)* (2011).

[30] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM '07)* (2007).

[31] KAWAGUCHI, M., LAHIRI, S. K., AND REBELO, H. Conditional Equivalence. Tech. Rep. MSR-TR-2010-119, Microsoft Research, 2010.

[32] KHARRAZ, A., ARSHAD, S., MULLINER, C., ROBERTSON, W. K., AND KIRDA, E. UNVEIL: A large-scale, automated approach to detecting ransomware. In *Proceedings of the 25th USENIX Conference on Security Symposium* (2016).

[33] KHARRAZ, A., ROBERTSON, W., BALZAROTTI, D., BILGE, L., AND KIRDA, E. Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'15)* (2015).

[34] KIRAT, D., AND VIGNA, G. MalGene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)* (2015).

[35] KIRAT, D., VIGNA, G., AND KRUEGEL, C. BareCloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (2014).

[36] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium* (2009).

[37] LAKHOTIA, A., PREDA, M. D., AND GIACOBAZZI, R. 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).

[38] LANZI, A., SHARIF, M., AND LEE, W. K-Tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS09)* (2009).

[39] LINDORFER, M., DI FEDERICO, A., MAGGI, F., COMPARETTI, P. M., AND ZANERO, S. Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)* (2012).

[40] LU, K., ZOU, D., WEN, W., AND GAO, D. deRop: Removing return-oriented programming from malware. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)* (2011).

[41] LUO, L., MING, J., WU, D., LIU, P., AND ZHU, S. 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).

[42] MENG, X., AND MILLER, B. P. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)* (2016).

[43] MING, J., PAN, M., AND GAO, D. iBinHunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)* (2012).

[44] MING, J., XIN, Z., LAN, P., WU, D., LIU, P., AND MAO, B. Replacement Attacks: Automatically impeding behavior-based malware specifications. In *Proceedings of the 13th International Conference on Applied Cryptography and Network Security (ACNS'15)* (2015).

[45] MING, J., XU, D., WANG, L., AND WU, D. 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).

[46] MING, J., XU, D., AND WU, D. Memoized semantics-based binary diffing with application to malware lineage inference. In *Proceedings of the 30th International Conference on ICT Systems Security and Privacy Protection (IFIP SEC'15)* (2015).

[47] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium of Security and Privacy (S&P'07)* (2007).

[48] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of static analysis for malware detection. In *Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07)* (December 2007).

[49] NG, B. H., AND PRAKASH, A. Exposé: Discovering potential binary code re-use. In *Proceedings of the 37th IEEE Annual Computer Software and Applications Conference (COMPSAC'13)* (2013).

[50] OH, J. W. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. Black Hat USA 2009, 2009.

[51] OKANE, P., SEZER, S., AND MCLAUGHLIN, K. Obfuscation: The hidden malware. *IEEE Security and Privacy 9*, 5 (2011).

[52] OREANS TECHNOLOGIES. Code Virtualizer: Total obfuscation against reverse engineering. `http://oreans.com/codevirtualizer.php`, last reviewed, 02/16/2017.

[53] PANDA SECURITY. 227,000 malware samples per day in Q1 2016. `http://www.pandasecurity.com/mediacenter/pandalabs/pandalabs-study-q1/`.

[54] PEWNY, J., GARMANY, B., GAWLIK, R., ROSSOW, C., AND HOLZ, T. Cross-architecture bug search in binary executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)* (2015).

[55] POULIOS, G., NTANTOGIAN, C., AND XENAKIS, C. ROPInjector: Using return oriented programming for polymorphism and antivirus evasion. Black Hat USA 2015, 2015.

[56] RAMOS, D. A., AND ENGLER, D. R. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)* (2011).

[57] ROUNDY, K. A., AND MILLER, B. P. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys 46*, 1 (2013).

[58] SCAIFE, N., CARTER, H., TRAYNOR, P., AND BUTLER, K. R. CryptoLock (and Drop It): Stopping ransomware attacks on user data. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS'16)* (2016).

[59] SCHRITTWIESER, S., KATZENBEISSER, S., KIESEBERG, P., HUBER, M., LEITHNER, M., MULAZZANI, M., AND WEIPPL, E. Covert Computation: Hiding code in code for obfuscation purposes. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS'13)* (2013).

[60] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)* (2007).

[61] SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Automatic reverse engineering of malware emulators. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (S&P'09)* (2009).

[62] SHI, Y., GREGG, D., BEATTY, A., AND ERTL, M. A. Virtual machine showdown: Stack versus registers. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE'05)* (2005).

[63] SIKORSKI, M., AND HONIG, A. Counterfeiting the Pipes with FakeNet 2.0. BlackHat EUROPE 2014, 2014.

[64] SMITH, J., AND NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[65] SMITH, T. F., AND WATERMAN, M. Identification of common molecular subsequences. *Journal of Molecular Biology 147*, 1 (1981).

[66] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)* (2008).

[67] SOOS, M., NOHL, K., AND CASTELLUCCIA, C. Extending SAT solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)* (2009).

[68] UGARTE-PEDRERO, X., BALZAROTTI, D., SANTOS, I., AND BRINGAS, P. G. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proceedings of the 36th IEEE Symposium on Security & Privacy* (2015).

[69] VMPROTECT SOFTWARE. VMProtect software protection. `http://vmpsoft.com`, last reviewed, 02/16/2017.

[70] VMRAY. VMRay Analyzer. `https://www.vmray.com/`, last reviewed, 02/16/2017.

[71] WANG, C., HILL, J., KNIGHT, J. C., AND DAVIDSON, J. W. Protection of software-based survivability mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks* (2001).

[72] WANG, T., WEI, T., GU, G., AND ZOU, W. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions on Information and System Security (TISSEC) 14*, 15 (September 2011).

[73] WANG, X., JHI, Y.-C., ZHU, S., AND LIU, P. Behavior based software theft detection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)* (2009).

[74] WARREN, H. S. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[75] XU, D., MING, J., AND WU, D. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)* (2017).

[76] XU, Z., ZHANG, J., GU, G., AND LIN, Z. AUTOVAC: Automatically extracting system resource constraints and generating vaccines for malware immunization. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS'13)* (2013).

[77] XU, Z., ZHANG, J., GU, G., AND LIN, Z. GoldenEye: Efficiently and effectively unveiling malwares targeted environment. In *Proceedings of the 17th International Symposium on Research in Attacks Intrusions and Defenses (RAID'14)* (2014).

[78] YADEGARI, B., AND DEBRAY, S. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)* (2015).

[79] YADEGARI, B., JOHANNESMEYER, B., WHITELY, B., AND DEBRAY, S. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)* (2015).

[80] ZHANG, X., GUPTA, R., AND ZHANG, Y. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)* (2003).

Appendix

Table 7: Examples: critical system calls/Windows API.

Object	Critical system calls/Windows API
File	NtCreateFile, NtOpenFile, NtClose
	NtQueryDirectoryFile, NtSetInformationFile
Registry	NtCreateKey, NtOpenKey, NtSaveKey
Memory	NtAllocateVirtualMemory, NtMapViewOfSection
	NtWriteVirtualMemory
Process	NtCreateProcess, NtOpenProcess , NtTerminateProcess
Thread	NtCreateThread, NtResumeThread, NtTerminateThread
Network	connect, bind, send, recv, gethostname
Desktop	CreateDesktop, SwitchDesktop, SetThreadDesktop
Other	LoadLibrary, GetProcAddress, GetModuleHandle

Table 8: Instructions with implicit branch logic.

Instructions	Meaning
CMOVcc	Conditional move
SETcc	Set operand to 1 on condition, or 0 otherwise
CMPXCHG	Compare and then swap
REP-prefixed	Repeated operations, the upper limit is stored in ecx
JECXZ	Jump if ecx register is 0
LOOP	Performs a loop operation using ecx as a counter

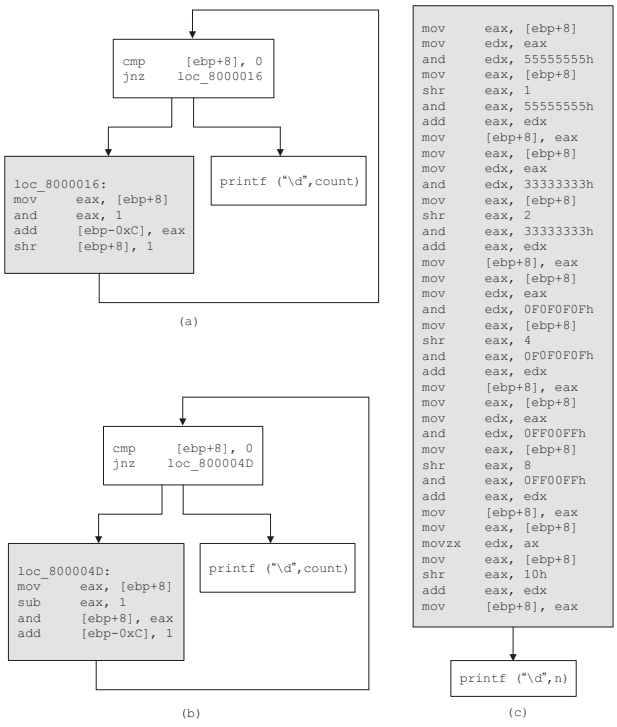


Figure 11: The disassembly code of three BitCount algorithms shown in Figure 3. The grey basic blocks represent the main loop bodies, which are not matched by "block-centric" binary diffing tools.

```
1  int isPowerOfTwo_1 (unsigned int x)
2  {
3     /* While x is even and > 1 */
4      while (((x % 2) == 0) && x > 1)
5          x /= 2;
6      return (x == 1);
7  }
8
9  int isPowerOfTwo_2 (unsigned int x)
10 {
11     unsigned int numberOfOneBits = 0;
12     while(x && numberOfOneBits <=1)
13     {
14         /* Is the least significant bit a 1? */
15         if ((x & 1) == 1)
16             numberOfOneBits++;
17         /* Shift number one bit to the right */
18         x >>= 1;
19     }
20     return (numberOfOneBits == 1);
21 }
```

Figure 12: Two different `isPowerOfTwo` algorithms check if an unsigned integer is a power of 2.

```
1  unsigned flp2_1 (unsigned x){
2      x=x|(x>>1);
3      x=x|(x>>2);
4      x=x|(x>>4);
5      x=x|(x>>8);
6      x=x|(x>>16);
7      return x-(x>>1);
8  }
9
10 unsigned flp2_2 (unsigned x){
11     unsigned y=0x80000000;
12     while(y>x){
13         y=y>>1;
14     }
15     return y;
16 }
17
18 unsigned flp2_3 (unsigned x){
19     unsigned y;
20     do{
21         y=x;
22         x=x&(x-1);
23     }while(x!=0);
24     return y;
25 }
```

Figure 13: Three different `flp2` algorithms find the largest number that is power of 2 and less than an given integer x.