Obfuscation Resilient Binary Code Reuse through Trace-oriented Programming

Junyuan Zeng[1], Yangchun Fu[1], Kenneth A. Miller[1], Zhiqiang Lin[1], Xiangyu Zhang[2], Dongyan Xu[2]
[1]Dept. of Computer Science, The University of Texas at Dallas, 800 W. Campbell Rd, Richardson, TX 75080
[2]Dept. of Computer Science and CERIAS, Purdue University, 305 N. University St, West Lafayette, IN 47907
[1]{jxz101120, yxf104020, kam103020, zxl111930}@utdallas.edu
[2]{xyzhang, dxu}@cs.purdue.edu

ABSTRACT

With the wide existence of binary code, it is desirable to reuse it in many security applications, such as malware analysis and software patching. While prior approaches have shown that binary code can be extracted and reused, they are often based on static analysis and face challenges when coping with obfuscated binaries. This paper introduces trace-oriented programming (TOP), a general framework for generating new software from existing binary code by elevating the low-level binary code to C code with templates and inlined assembly. Different from existing work, TOP gains benefits from dynamic analysis such as resilience against obfuscation and avoidance of points-to analysis. Thus, TOP can be used for malware analysis, especially for malware function analysis and identification. We have implemented a proof-of-concept of TOP and our evaluation results with a range of benign and malicious software indicate that TOP is able to reconstruct source code from binary execution traces in malware analysis and identification, and binary function transplanting.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and re-engineering*; D.2.m [**Software Engineering**]: Miscellaneous—*Reusable Software*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Binary code reuse, trace-oriented programming, taint analysis, dynamic decompilation

1. INTRODUCTION

Binary code reuse involves extracting selected pieces of code from an application binary, recompiling and linking them with other components, and producing a new software program. Binary code reuse is desirable in many security applications such as malware code inspection and classification, legacy binary program retrofitting, security function transplanting, and source code recovery.

While decompilation [16, 17] has long been adopted for source code recovery and hence binary code reuse, it is based on static analysis and has limitations. For example, a state-of-the-art decompilation tool, Hex-Rays [2] is able to generate C code that is semantically equivalent to the original binary code in terms of execution effects. It was used to recover the Stuxnet source code in 2011 [19]. However, the source code generated by Hex-Rays may be unsafe. More specifically, it does not fully recover non-trivial indirect jump targets or function pointers, due to its static analysis nature; nor does it deal with binaries with *obfuscation*, such as instruction aliasing [3] or encrypted binary code case (e.g., those obfuscated by packers). While there are recent advances for improving static decompilation, such as semantic-preserving structural analysis [51], some challenges still remain. Moreover, the recovered source code very often may not be directly compilable.

Recent efforts – notably BCR [9] and Inspector Gadget [38] – can extract part of a malware binary for reuse or inspection. However, they are also not safe as they rely on incomplete dynamic analysis without a reliable mechanism to remedy the partial information acquired from the dynamic execution. Virtuoso [24] is a technique that extracts execution traces and translates them to executable python programs that could run outside the platform. In addition to similar limitations, it requires special runtime support – a python interpreter – to execute and often incurs high performance overhead (up to hundreds of times of slowdown). VMST [29] addresses Virtuoso's limitations but it only supports kernel functions and requires a heavy-weight dual-VM architecture [30].

This paper presents a new binary code reuse framework called *trace-oriented programming* (TOP) which is built on dynamic binary analysis. The basic idea of TOP is that, given a binary (possibly obfuscated), through dynamic analysis of its execution, we collect instruction traces and translate the executed instructions into a high level program representation using C with templates and inlined assembly (for better performance). Such a program representation can be directly compiled and linked with other code to produce new software. Unlike static analysis-based decompilation, TOP is based on dynamic analysis. Therefore, it has better resilience against obfuscation [18] and it does not require any binary points-to [8, 39, 48] or alias analysis [22, 7].

A by-product of TOP is the ability to instrument the newly generated code with additional guards, thereby gaining the ability to mitigate the incompleteness caused by dynamic analysis. In particular, since TOP targets binary code reuse, where the recovered code needs to run independently or be linked with other programs, it

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright 2013 ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516664.

must ensure that the newly generated reusing code (i.e., the reusable software component) reproduces the same behavior with the traces and flag exceptions if the behavior is not consistent with the traces. In other words, there is no gray area in the newly generated code, with its behavior well-defined by the traces. By instrumenting the code while reconstructing its source code, TOP is able to achieve such behavior consistencies.

We have implemented a TOP prototype and evaluated it with a range of benign and obfuscated binary programs. Our evaluation results show that we can directly compile the reconstructed source code into new binaries, and run them correctly with the functionality we have traced. Using TOP, we demonstrate two compelling applications: (1) malicious code inspection and identification, and (2) security function transplanting.

In summary, this paper makes the following contributions:

- We present trace-oriented programming (TOP), a new framework to reconstruct program source code from execution traces. Unlike decompilation that statically transforms a piece of binary code, TOP dynamically translates it with more runtime information and directly generates reusable software components.

- We devise a number of enabling techniques for TOP such as program control structure recovery, address symbolization, safety instrumentation, and instruction-to-C translation. These techniques work together to recover the source code from dynamic execution of a binary, and ensure that the recovered code has well-defined behavior consistent with the execution traces.

- We propose a systematic technique to symbolize the function pointers that are initialized and in global memory regions. To the best of our knowledge, this has not been proposed in any existing static decompilation technique including Hex-Rays.

- We have built our prototype systems for Windows and Linux platforms, and tested them with over one hundred pieces of benign and obfuscated binary code. In addition, we apply TOP to two security applications: malware analysis (e.g., unpacking) and identification, and security function transplanting.

The rest of the paper is structured as follows. In §2, we motivate TOP, and give an overview of our key techniques. The detailed design of TOP is presented in §3, followed by the evaluation in §4. Then in §5, we demonstrate the application of TOP in four usage scenarios. We discuss limitations and future work in §6, and review and compare with related work in §7. Finally, §8 concludes this paper.

2. BACKGROUND AND OVERVIEW

2.1 Goals and Properties

Targeting binary code reuse, TOP differs from the traditional static analysis-based decompilation techniques in that it performs *dynamic decompilation of program traces*. The salient properties of TOP are as follows:

- **Resilient against Obfuscation.** For intellectual property protection or anti-malware analysis purpose, many binary programs are obfuscated. Obfuscation techniques [18] range from instruction aliasing, garbage code insertion, register reassignment, instruction substitution, to binary code encryption and packing [35], and even virtualization-based obfuscation [53]. However, no matter how complicated the obfuscation technique is, the obfuscated program has to be executable. From its execution traces, we can recover the program's source code, although it may not be exactly the same as the original code (e.g., those obfuscated with virtualization), and reuse it in new programs.

- **Free from Point-to and Alias Analysis.** It is challenging to perform point-to or alias analysis [8, 39, 48, 22, 7] statically. However, since TOP is based on dynamic execution, the point-to relations are exercised directly, hence there is no need to perform complicated point-to analysis to figure out pointers' targets.

- **Concrete Instead of Abstract Values.** Similar to concolic testing [33, 52] (a mix of symbolic and concrete execution), concrete values are sometimes preferred in some reuse scenarios. By using TOP, values of the variables accessed during the program's execution can be observed. When generating the new source code, some arguments can be replaced with concrete values to avoid certain complicated tasks, such as environment setup and pre-condition computation; and a function pointer can be replaced with a concrete function call if the pointer always points to a specific function in the trace.

2.2 Challenges

It is a well-established approach to translate machine code back to human readable disassembled code (which is called disassemble). However, from disassembled code to high-level code (i.e., decompilation [16, 17, 46]), there is no standard approach and no significant breakthrough in the past few years for native code (although decompilation is more successful for other low-level code such as Java-bytecode [43]).

TOP is built atop dynamic binary instrumentation (DBI) which can be used to generate execution traces. However, the traces from DBI-tools (e.g., PIN [41] and QEMU [4]) cannot be reused directly, and we must solve the following challenges:

- **Control Structure Identification.** Normally a trace is a sequence of instructions executed by the CPU, and there is no explicit control structure inside. For example, the loop is unrolled, and callers and callees become sequential in the trace. Therefore, we have to identify the program control structures such as loops and function calls for the C code being generated.

- **Type Classification of Literal Values.** The literal value (i.e., the immediate value) in an instruction can be associated with different types, such as a global or read-only data address, a function pointer or a constant. If the literal value refers to a memory access or a function pointer, it has to be converted to the address which is associated with the generated C code instead of the original binary code. Therefore, we have to develop techniques to precisely differentiate the types of the literal values and symbolize them.

- **Safe Reuse.** Based on dynamic analysis, TOP faces the code coverage challenge. However, our goal is to make the non-exhaustively executed code reusable and ensure that the recovered code is safe and consistent with the traces. For example, as shown in Fig. 2, if we do not have any safety check in the partially executed code, the original semantics of the program may get violated. Thus, we have to develop techniques to ensure the safety of the recovered source code.

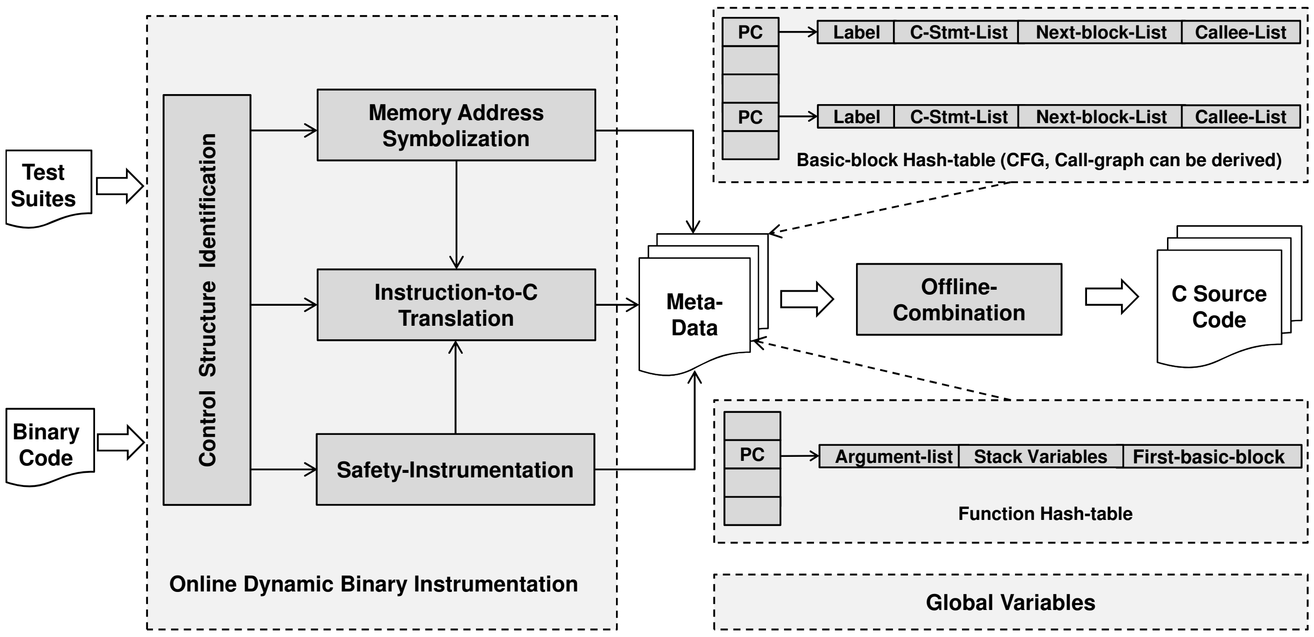

Figure 1: Overview of TOP Framework.

```
1 if (year_of_service > 10) {
2     if (salary < 100000.0)
3         salary = 100000.0;
4     else
5         salary = salary*1.02;
6 }
```

(a) Original Source Code

```
1 if (year_of_service > 10) {
2     if (salary < 100000.0)
3         salary = 100000.0;
6 }
```

(b) Naively Recovered Source Code from Trace

Figure 2: Partial code recovery issue faced by TOP. Assume the provided input is `year_of_service = 15` and `salary = 80,000`. An unsafe code recovered includes lines 1, 2, 3, and 6 of the original program.

2.3 Overview

An overview of TOP is presented in Fig. 1. The input to TOP is the application binary code and a test suite; and the output is the executed, modularized, and reusable components with C source code representation, which can then be directly compiled and linked to generate new programs.

There are five key components in TOP, four of which are designed using online DBI. More specifically, through dynamically monitoring executed instructions, we will identify the control structure of the code (§3.1) such as function calls, loops, and branches; discover the literal value types and relocate them with symbols (if necessary) such that they can be recompiled (§3.2); collect the control flow graph (CFG) for safety instrumentation (§3.3); and translate instructions using inline assembly for C code (§3.4). Finally, we will have all the meta-data necessary to reconstruct the C code. The offline-combination component (§3.5) will combine multiple runs if necessary and emit the final C code.

3. DETAILED DESIGN

3.1 Control Structure Identification

The typical control structures in binary code include sequence, selection or conditional branch (`jcc`, namely jump if condition is met; there are 77 such instructions in x86 [1]), and repetition (i.e., loop). However, loop is essentially a special case of conditional branch which has a backward edge in the control flow graph (CFG). In other words, at the binary level, we only see instructions being executed either sequentially, or with a control flow transfer (including explicit ones such as `jcc/jmp/ret/call` and implicit ones such as `push/ret`).

Our goal is to translate binary code (more precisely the dynamically disassembled code) into C code that can be recompiled. In fact, control structures are already encoded in the binary, especially in the program's CFG, where each node represents a basic block (BB) in which instructions are executed sequentially; and a directed edge represents the control flow transfer (such as `jcc/jmp`).

CFG is not only crucial to *control structure identification*, but also important for our *safety instrumentation* technique (§3.3) with the goal of tolerating incomplete execution (i.e., only one of the two branches of a conditional gets executed and dynamically decompiled). Therefore, the first step of TOP is to dynamically build a program's CFG based on executed instructions.

Dynamic CFG Construction. Since we target programs that may be obfuscated or metamorphic, we cannot rely on static analysis (e.g., static disassembler). Hence, our key idea for CFG construction is to leverage the exercised instruction addresses to safely connect the BBs (with their successors and predecessors) during the program's execution. In particular, we focus on handling metamorphism that may cause instructions being dynamically modified or replaced (e.g., those obfuscated by packers). The procedure for dynamic CFG construction is presented in Algorithm 1.

For a given executing instruction with address PC, if the instruction is `jcc/jmp/call` (line 4 -11), we will first get the PC of the next instruction (line 5) and retrieve its current layer. The reason to introduce a layer (an unsigned integer) for each PC is to track the version of the newly generated code, and use $\langle PC, layer\rangle$ as the unique index to access a basic block hash table (BBHT). Without layers, we cannot have a one-to-one mapping because the same address can be overwritten and one PC can be mapped to different

```
1: Input: $PC$, the current executing instruction address; $Shadow$: the shadow
   memory of the accessed program address space, which stores the current memory
   overwriting layers.
2: Output: Partial CFG stored in the basic block hash table (BBHT)
3: DynamicBinaryInstrumentation ($PC$, $BB$) {
4:     if (Inst($PC$) $\in$ {jcc, jmp, call})
5:         $next\_pc$ $\leftarrow$ Next({jcc, jmp, call})
6:         $layer$ $\leftarrow$ GetShadowLayer($Shadow$, $next\_pc$)
7:         $N$ $\leftarrow$ GetBB($next\_pc$, $layer$);
8:         $T$ $\leftarrow$ Set of exercised successor of PC;
9:             if (N $\notin$ $T$)
10:                Insert N to BB's successor in BBHT;
11:                Insert BB as N's predecessor in BBHT;
12:    else if (Inst($PC$) $\in$ {ret})
13:        $next\_pc$ $\leftarrow$ Stack($ESP$),
14:        $layer$ $\leftarrow$ GetShadowLayer($Shadow$, $next\_pc$)
15:        $N$ $\leftarrow$ GetBB($next\_pc$, $layer$);
16:        $T$ $\leftarrow$ Set of exercised successor of PC;
17:            if (N $\notin$ $T$)
18:                Insert N to BB's successor in BBHT;
19:                Insert BB as N's predecessor in BBHT;
20:    else if (Inst($PC$) $\in$ {Memory Write})
21:        Increase the layer by one in the shadow memory.
22:    Append $PC$ to $BB$ if this $PC$ has not been visited;
23: }
```

Algorithm 1: *Dynamic CFG Construction*

instructions at different times (e.g., in different unpacking phases if the program is packed by multiple packers). Next, we will retrieve the next BB (i.e., N) of $\langle next\_pc, layer\rangle$ (line 7); if N does not exist, `GetBB` will create one. After that, it retrieves all the successors of current PC, and updates the successor and predecessor of the current BB (line 10-11). Handling of `ret` is similar, except that we have to fetch its next PC from the stack (line 13). Also, for memory write instructions, since we need to track the layer of a memory address, we retrieve the current layer for the overwritten address and increase it by one. Then whenever an instruction is fetched from the memory, we can retrieve its layer from the shadow memory. At the end of the execution of the current instruction, we will append the PC to the current BB if it is visited for the first time (line 22). Note that our BBHT contains all the control flow transfer information of the executed instructions, all of our following components will leverage the information collected in BBHT as illustrated in Fig. 1.

3.2 Memory Address Symbolization

During program execution, many instruction and memory data addresses become concrete. For instance, the target of a control flow transfer instruction is a concrete instruction address, and the target of a memory access operation is also a concrete address. However, in the recovered code, we often cannot directly use these concrete addresses. Instead, we have to symbolize and relocate them. We call this procedure *Memory Address Symbolization*.

In general, we have to perform (1) instruction operand symbolization and (2) global data symbolization. We do not need to symbolize heap or stack memory addresses because they are dynamically allocated and intrinsically relocatable. For instruction operand symbolization, we need to rewrite a concrete address or literal value with a symbolized address (such as a label). For instance, as shown in the first example in Table 1, we need to rewrite the target address of direct `call` with the address of `func_dest_addr`; and the destination address of `je` with label `L_dest_addr`. For global data symbolization (the third and fourth examples in Table 1), we need to rewrite the instructions that use concrete addresses of the global variables. Also, we need to rewrite the initial value stored in a global variable if the value itself is an address of a global variable or a function pointer.

Unfortunately, memory address symbolization is challenging. This is because literal values are widely used in many instructions. Many of them are not addresses even though some of them look like so. Only from the later data use of a value, we can infer if it is a memory address, a pointer (with its content being a memory address), or a double pointer (from a two-layer pointer dereference). Also, there could be indirect `call`/`jmp` or even `ret` (as shown in the second example in Table 1). We have to back track to decide whether their target operand is directly or indirectly derived from any literal value representing a global address. To this end, we adopt a data dependence tracking (i.e., taint analysis [15, 47, 20]) algorithm to resolve the instructions whose operands need rewriting, and the global memory addresses that need to be updated with new symbols. In the following, we present the detailed design of our algorithm.

3.2.1 *Taint Sources*

The goal of our algorithm is to (1) pinpoint the instructions that have a memory operand that needs to be symbolized and (2) pinpoint every global data location that stores an initial function pointer or a global data pointer which need to be symbolized. Therefore, *our taint source contains the instruction address (PC) whenever an instruction involves a literal value operand, or a data value that appears to be a global pointer or a function pointer*. For either case, we are able to determine the operand or the data value is indeed a global/function pointer by observing that it gets dereferenced at a later point. We replace the origin of the operand, which is indexed by the taint source using its PC, with a symbol; or the origin of the memory data indexed by the taint source using its value with a symbol. We will use the examples in Table 1 to illustrate how our analysis works.

Consider the third example in Table 1 (line 8-11 in the third column). When line 8 gets executed, we will assign the shadow record for `eax` as S[`eax`]=8 (PC) because "`mov eax, 0x424a38`" has a literal value. Instruction address 8 will be used later to update the source operand 0x424a38 at location 8 if the taint sink point (discussed below) indicates that `eax` stores a memory address. When line 9 "`mov ecx, DWORD PTR [eax]`" is executed, it is a taint sink point as it dereferences the value stored in `eax`. We hence know that `eax` stores a global memory address. We then update the operand at address 8, and generate an address-symbolized instruction "`mov eax, OFFSET global_data+0x8`" as shown in line 8 in the last column in Table 1. Also, this instruction denotes a new taint source as the value stored in `eax`=0x424a38 is 0x424a34 that appears to be a pointer pointing to the global area. Hence, the memory address of the source operand is assigned to the shadow record of `eax`, S[`ecx`]=0x424a38.

3.2.2 *Taint Propagations*

Much like all other taint analysis, the taint record gets propagated along with data movement instructions, and data arithmetic (because of pointer value computation) instructions. For instance, along with data movement instruction at line 10 "`mov edi, ecx`", S[`ecx`]=0x424a38 will be propagated to `edi` (i.e., S[`edi`]=0x424a38).

3.2.3 *Taint Sinks*

A taint sink is an instruction point that can reveal the type of the operand: whether a literal value is a global memory address or relative address for `call`/`jmp` target; or whether the involved memory operand is a pointer. Therefore, nearly all instructions are possible taint sinks. When a taint sink instruction is executed, we will update the assembly code depending on whether it involves instruction operand symbolization or global data symbolization.

Case I: Instruction Operand Symbolization. There are many instructions that take literal values. For some instructions we can

Examples		Original Assembly Code	Address Symbolized Code

Instruction Operand Symbolization	Direct Symbolization	`1:  je   0x401175`
		`2:  call 0x401028`
		`3:  mov  DWORD PTR 0x424a30, 0x1`
			`1:  je   L_0x401175`
			`2:  call func_0x401028`
			`3:  mov  DWORD PTR [global_data+0x0], 0x1`

	Indirect Symbolization	`4:  push 0x401058`
		`5:  ret`
		
		`6:  mov  eax, 0x409200`
		`7:  call eax`
			`4:  push OFFSET L_0x401058`
			`5:  ret`
			
			`6:  mov  eax, OFFSET func_0x409200`
			`7:  call eax`

Global Data Symbolization	Implicitly Initialized Global Data	`8:  mov  eax, 0x424a38`
		`9:  mov  ecx, DWORD PTR [eax]`
		`10: mov  edi, ecx`
		`11: mov  ebx, DWORD PTR [edi]`
		
		`12: call DWORD PTR 0x424a3c`
			`8:  mov  eax, OFFSET global_data+0x8`
			`9:  *(int*)(global_data+0x8) = global_data+0x4`
			`10: mov ecx, DWORD PTR [eax]`
			`11: mov edi, ecx`
			`12: mov ebx, DWORD PTR[edi]`
			
			`13: *(int*)(global_data+0xc) = func_0x40102d`
			`14: call DWORD PTR [global_data+0xc]`

	Explicitly Initialized Global Data	`13: mov  DWORD PTR 0x424a38, 0x424a30`
		`14: mov  ecx, DWORD PTR 0x424a38`
		`15: mov  DWORD PTR [ecx],0x4`
		
		`16: mov  DWORD PTR 0x424a3c, 0x401032`
		`17: call DWORD PTR 0x424a3c`
			`15: mov DWORD PTR [global_data+0x8],OFFSET global_data+0x0`
			`16: mov ecx, DWORD PTR [global_data+0x8]`
			`17: mov DWORD PTR [ecx],0x4`
			
			`18: mov  DWORD PTR [global_data+0xc],func_0x401032`
			`19: call DWORD PTR [global_data+0xc]`

Table 1: Examples of Memory Address Symbolization.

immediately infer that the operand is a memory address (we call it direct symbolization). For the others we must infer based on later data use (we call it indirect symbolization). Therefore, we have the following two strategies:

- **Direct Symbolization.** Direct symbolization applies to the direct control flow-related instructions (i.e., `call`/`jmp`/`jcc`). In particular, if the operand of such an instruction is a literal value, we will directly symbolize it. For instance, as shown in line 1-2 of the first example in Table 1, we will directly symbolize `0x401175` with a label `L_0x401175` for the `je` instruction; and `0x401028` with `fun_0x401028` for the `call` instruction. Direct symbolization is critical for unpacking. Usually the last step when finishing unpacking is through a control flow transfer instruction (`jmp/call/ret`). If we do not symbolize the target address of the last instruction, TOP would only capture the code for unpacking and the new code's semantics will not be consistent with our traces.

- **Indirect Symbolization.** At the data use point, after we determine the literal value to be a symbol address, we will look for the target instruction based on the PC in the taint record and rewrite the symbol. For instance, as shown in the second example in Table 1, at line 7 when "`call eax`" gets executed, we can infer that the value stored in `eax` is actually a function entry address. Since the taint record of `eax` is S[`eax`]=6, we will then rewrite the operand of the instruction at PC=6 and symbolize the literal value as `fun_0x409200`. For the return instruction, we will also check the taint record of the operand that is from the top of the stack. If it is tainted, we will update the target instruction as well. For instance, because line 4 "`push 0x401058`" and line 5 directly return, fetching value 0x401058 from the stack (this instruction is actually a case of control flow obfuscation), we will rewrite the operand of the `push` instruction as `L_0x401058`. Note that for a normal `call` instruction, unlike in this case, we will not taint its return address on the stack.

Case II: Global Data Symbolization. Compared with instruction operand symbolization, global data symbolization is more complicated and it requires not only translating the concrete global address into symbolic address, but also translating the *pointed* data stored in the global variable as symbolic. There are also two cases for global data symbolization depending on whether the global data accessed is implicitly initialized or explicitly initialized (or redefined).

- **Implicitly Initialized Global Data.** A global pointer (either data pointer or function pointer) could be initialized by a compiler, often with no more update during program execution. In this case, we need to symbolize the stored pointer value explicitly (with an assignment statement to be inserted at the beginning of the recovered code); otherwise the translated program will crash.

  Take the instruction at line 11: "`mov ebx, DWORD PTR [edi]`" as an example (the third row in Table 1). When this instruction is executed, we can infer that `edi` stores a pointer. At this moment, S[`edi`]=0x424a38, which indicates that we are dereferencing a memory address pointed to by the value stored in 0x424a38. In other words, we know that 0x424a38 actually stores a pointer and this pointer is implicitly initialized (because there is no other instruction to define this memory address). Therefore, we have to explicitly translate the content stored in 0x424a38. In this case, it happens to be 0x424a34. That is why we add an explicit assignment statement at line A in the last column (the address of this instruction depends on the final recovered code but it needs to inserted at the beginning of the recovered code).

  This example also indicates that *we need to track data-def and data-use of the memory cell* (details are elided since it is a standard algorithm). If there is no data-def for a particular memory cell, it will be a case of implicit data initialization. Similarly, for line 12 "`call DWORD PTR 0x424a3c`", we will add a function pointer initialization statement at line B in the last column, and that function pointer happens to point to a function at address 0x40102d. Similar to the statement in line A, this assignment statement should be placed at the beginning of the recovered code.

- **Explicitly Initialized Global Data.** If a global function pointer is explicitly redefined (e.g., memory `0x424a38` in line 13), its handling will be simpler. We only need to symbolize the address of the global variable, without adding any explicit assignment statement (e.g., lines A and B in Table 1) for the global pointer variable. Instead, we will use the original program's code to dereference the memory. For instance, in the fourth example in Table 1, we only need to symbolize the operand at line 13 for memory addresses 0x424a38 and 0x424a30, and similarly at line 14 for memory address 0x424a38.

Again, we do not need to symbolize program heap and stack addresses. The main reason is that those addresses are dynamic hence the program code itself will initialize them and dereference them correspondingly. It is also important to note that *Memory Address Symbolization* is new in TOP; and none of the existing decompilation techniques, including Hex-Rays, has solved this problem, especially for global initialized pointer data.

3.3 Safety Instrumentation

TOP features the assurance of safety while leveraging the precise but incomplete dynamic analysis. To achieve this goal, we need to ensure that *the recovered code either behaves the same way as the original binary, or must throw predefined runtime safety exceptions.* We call this property *reuse safety*.

There are two root causes for safety violation: (1) A conditional jump (i.e., `jcc`) may not have both branches covered (as shown in Fig. 2) and (2) an indirect `jmp/call` may not have all its targets covered.

- **Handling Missing Conditional Branches.** If only one of the branches is executed in a `jcc` instruction, TOP will generate a piece of exception handling code that will print a warning message and exit the program if the control flow of the recovered code falls through the missing branch. This warning message will be used to debug and refine the extracted code. In particular, we can leverage the input that induces the warning as an additional input to increase code coverage.

- **Handling Indirect Jumps.** Similar to conditional branches, indirect jumps/calls can lead to safety issues due to incomplete path coverage. Two types of indirect jump/call – `jmp/call` register and `jmp/call` memory – are considered. TOP translates an indirect jump/call into a list of conditional jumps with the possible values of the symbolized indirect jump/call operands as conditions. To guarantee safety, an assertion is added to the beginning of the conditional jump list to check if the target is among the symbolized addresses. In other words, our code creates a white list of the symbolized target addresses; any unknown (new) target will be captured and thrown out.

With the safety instrumentations above, we ensure that, when the recovered code is executed again, it will follow the original program's semantics or throw exceptions. Any attempt to execute a control flow path not executed before will be warned.

3.4 Instruction-to-C Translation

After we collect the program's CFG (§3.1), relocate concrete memory addresses to symbols (§3.2), and instrument the extracted program to ensure safety (§3.3), the next step is to translate the low-level instructions into C code. Since our targeted usage scenario is binary code reuse, users may not be interested in the implementation details of the extracted components but only their functionality – for example, a user may not care how a cipher function is implemented. Therefore, in the generated C code, we largely use the inlined assembly and their operands will be updated with symbolized addresses if any.

Also, since our goal is to recover the extracted code in the form of source code, we could generate the source code for instructions at various levels, including user level, library level, and even kernel level (if we use virtual machine-based dynamic instrumentation). However, we are most interested in the user-level code (because library code already exists). As such, the recovery process needs to stop when encountering a well-defined API. We also need to dump the code from the beginning of the execution by default, or from the entry to the exit points specified by the user.

API Resolution. The goal of API resolution is to instruct our analysis to stop further tracing when entering the body of a library function and, at the same time, to use the well-defined interface and the API symbols to generate the function call (e.g., `malloc`, `printf`, `recv`). While there might be API-obfuscation in the binary code (e.g., to hide malware behavior from static analysis), our dynamic analysis can discover such obfuscation.

At a high level, our API resolution technique turns off address space randomization (which can be done by TOP users because they control the execution), and identifies the starting address for each library call. At runtime, we check whether the PC of an instruction matches any pre-defined API's PC. For Linux, we extract all the APIs in `glibc` and other dependent libraries if any, and resolve each API's symbol, arguments, and the starting PC.

For Windows binary, if DLLs are loaded at the virtual addresses specified as the image base address in a DLL's PE header[1], we will create a lookup hash table that contains all the virtual addresses of each exported API function from all known DLLs. If the calling target address can be found in the lookup table, we will retrieve the API calling context (i.e., name and arguments). If the DLLs are loaded to a nonstandard base address by system calls to explicitly map them to a different address space, we will look for specific calls to `NtOpenSection` which identifies the DLL name, and calls to `NtMapViewOfSection` which provides the DLL's base address. We then use the base address of this DLL to add those API addresses inside it.

Instruction Translation. Since TOP cares about code functionality more than code readability, we use a straightforward but effective inlined assembly rewriting approach to generate the corresponding C code. In particular, we traverse the BBHT (§3.1), starting from a user-specified address or a default address. For each executed instruction, if an absolute address is used, we will replace it with our symbolic address. If there is a missing branch that is not executed, we will instrument it with a safety check. If there is a global data access, we will associate it with an index (global variables are mapped to a big array) and initialize it with value in the memory. A control flow target will be rewritten as a specific symbolic label. If we encounter the end instruction, we will dump the C source code. Note that the C code is not just inlined assembly because we do recover information that is lacking in assemblies, such as variable types and function interfaces.

3.5 Offline Combination

To enable the *combination of multiple runs* of a binary, we also design a feature that serializes our in-memory meta-data to disk files, and then use the *offline-combination* component to combine the multiple runs for larger coverage. For offline combination, all

[1]Since the bases of standard Windows DLLs do not conflict with each other, the loader and linker can load all DLLs at their specified base virtual addresses.

Benchmark Programs			Online Phase of TOP			Offline Combination		Recovered Source Size (KLOC)	Consistent w/ Traces?
Category	Binary Programs	Assembly Code (LOC)	$\#Symbolized$ Addresses	$\#Safety$ Checks	Time (sec)	Test Cases	KLEE/TOP Coverage(%)		
Hash	sha512sum	25331	133	39	0.25	35	59.53	15.4	✓
	sha384sum	25331	133	39	0.25	45	59.74	15.4	✓
	sha256sum	9129	122	36	0.09	36	60.70	5.5	✓
File	vdir	18676	788	196	0.71	65	26.93	4.8	✓
	dir	18676	507	116	0.19	70	25.67	4.8	✓
	ls	18676	501	116	0.19	68	25.81	4.8	✓
Privilege	chown	10099	216	88	0.05	58	39.53	4.1	✓
	chmod	9811	208	81	0.02	54	29.77	2.9	✓
	chgrp	9436	196	80	0.04	50	31.08	2.9	✓
Disk	du	14887	487	173	0.06	66	38.43	5.7	✓
	df	9100	290	74	0.11	58	32.92	2.9	✓
	sync	4221	45	9	0.21	18	39.26	1.6	✓
Process	sleep	4541	85	25	0.08	58	42.05	1.9	✓
	kill	4935	83	31	0.04	37	38.71	1.9	✓
	nice	4680	43	11	0.05	47	41.17	1.9	✓
Environment	who	6097	363	147	0.20	27	36.34	2.2	✓
	env	4292	53	13	0.09	22	33.01	1.4	✓
	printenv	4290	50	9	0.09	12	24.08	1.0	✓
Utility	od	10172	245	61	0.09	119	53.42	5.5	✓
	pr	7611	480	130	1.01	79	40.03	3.0	✓
	wc	5959	210	45	0.24	45	44.47	2.7	✓

Table 2: Evaluation results with top-3 binary programs in 7 categories (in terms of recovered source code size).

the serialized meta-data are loaded into memory, and the CFG in the multiple meta-data is traversed. Whenever there is a path that is executed in one run but not in another, we will eliminate the safety instrumentation code and combine the two runs. After the combination, given the starting PC of a target function to reuse, TOP will traverse the BBHT, translate the instructions, and dump the source code reachable from the target function.

4. EVALUATION

We have implemented a proof-of-concept prototype of TOP. To handle obfuscated Windows binary code, we choose QEMU-1.0.1 [4] as the underlying dynamic binary instrumentation engine. We perform virtual machine introspection (VMI) [31] to inspect the target process and thread, intercept all executed instructions, collect context information, and resolve and store the information in the BBHT. To show the effectiveness and generality of TOP, we have implemented process/thread introspection and API resolution for both Windows and Linux. It is important to note that the five key components of TOP are *generic*, hence there is no need to customize for Windows and Linux.

More specifically, TOP needs to trace binary execution at thread level. To identify thread-level context, our introspection will use both process ID (by traversing the corresponding kernel data structure) and kernel stack pointer (with the lower 12 bits masked). This is because each thread will have a unique kernel stack (besides user level stack). Also, many programs create new processes. TOP tracks process creation by inspecting relevant system calls, and performs binary code translation and isolation for all child processes.

In this section, we present the results of evaluating TOP with a large number of legacy Linux binary programs (§4.1) and obfuscated Windows binaries (§4.2). The evaluation is performed on a machine with Intel Core i-7 CPU with 8GB physical Memory. The host Linux platform runs kernel-2.6.38; and the guest OS is Ubuntu-11.04 or Windows XP SP3.

4.1 Evaluation with Legacy Linux Binaries

Unlike the large pool of obfuscated binaries on Windows, much fewer obfuscated binaries exist on Linux. Hence our evaluation of TOP for normal, un-obfuscated binaries is mainly performed on Linux. We use the coreutils-6.11 package as our benchmark suite. Our goal is to use TOP to generate C code from execution traces, and further recompile them to generate customized binaries. We compare the functionality of the new binaries with the original ones for effectiveness evaluation.

Effectiveness. There are in total 100 programs in coreutils-6.11. Most of them are single process except three multi-threaded programs, `sort`, `mkdir` and `ginstall`. TOP can detect the thread-level control flow correctly. In our experiment, we compile them with "`gcc -O2`", strip out their symbols, and run each of them to generate the corresponding source code from the trace.

When running these programs, we do not provide explicit command line option (e.g., we just type `ls`), unless we have to provide one such as for `cat`. For those that require files as input, we provide files with a size of 1KB. Next, we run the the recompiled binaries with the same option for tracing, and compare the output to test whether we retain the correct semantics of the original binary with the given inputs.

As expected, all 100 benchmark programs run successfully, and consistently generate the same result as their original counterparts for the same input. If we enter any other command line options, the new programs will generate exceptions and exit gracefully. Note that in such a case, the user can choose to further generate a more complete version of the recovered program using the exception-inducing input. For space constraint, we classify these 100 programs into seven categories and rank the source code size (in terms of LOC) generated for each program, as shown in the first column of Table 2. For each category, we report the top 3 programs in the 2^{nd} column of Table 2 for detailed presentation of our results. The 4^{th} column reports the number of symbolized memory addresses, and the 5^{th} reports the number of safety checks TOP added. These results reveal more details about the internal operations of TOP as well as the program-specific characteristics in these aspects. We observe that all programs require hundreds of symoblized addresses. For safety checks, the number varies across programs.

Performance Overhead. There are two kinds of performance overhead: (1) overhead of TOP tracing binary execution and generating

Obfuscated Samples	#bytes	Obfuscation Techniques	Anti-Disas.	Anti-Debug	Anti-VM	#Symbolized	#Safety	#LOC	Coverage	Consistent?
garbage_bytes.exe	1536	Adding garbage bytes	✓			3	1	38	71.4%	✓
program_control_flow.exe	1536	CFG Obfuscation (push/jmp)	✓			3	0	30	100%	✓
pushret.exe	1536	CFG Obfuscation (push/ret)	✓			3	0	40	100%	✓
call_trick.exe	1536	CFG Obfuscation (call/push/ret)	✓			8	0	71	100%	✓
middle_instruction.exe	1536	Instruction Aliasing	✓			10	1	78	100%	✓
Win32.Bamital.exe	22016	Encryption Packer	✓			303	37	1264	-	✓
Virus.Win32.Adson.exe	5632	Compression Packer	✓			60	6	739	-	✓
hardware_bp.exe	1536	Hardware Breakpoint		✓		12	4	78	70.8%	✓
heapflags.exe	1536	Heap Flags Detection		✓		9	1	59	81.8%	✓
instr_counting.exe	1536	Instruction Counting		✓		12	3	136	65.6%	✓
ntglobal.exe	1536	PEB NtGlobalFlag		✓		9	1	71	81.8%	✓
peb.exe	1536	IsDebuggerPresent		✓		9	1	59	81.8%	✓
rdtsc.exe	1536	RDTSC Instruction Timing		✓		8	1	75	90.0%	✓
softice.exe	1536	Softice Interrupt		✓		8	1	75	83.3%	✓
software_bp.exe	1536	Soft Breakpoint Detection		✓		10	1	77	91.7%	✓
ss_register.exe	1536	SS Register		✓		9	1	83	88.2%	✓
anti-vm_in_instruction.exe	1536	Anti-Vmware IN Instruction			✓	7	0	59	100%	✓

Table 3: Evaluation results with 17 obfuscated binary programs.

new source code and (2) overhead of new software generated compared with the original software.

For the 100 programs, TOP on average runs about 0.2 second to generate the new program from a single trace. To further study performance overhead, we present the TOP runtime (incurred by its four online components) in the 6^{th} column in Table 2. For each program, it takes less than one second for TOP to finish, except utility `pr` that takes 1.01 seconds. For each new program generated, we recompile and run it using the same arguments as those used for tracing. Since the translated code is almost identical to the original assembly code, the new program incurs negligible performance overhead, thanks to the use of inlined assembly.

Coverage. Recall that TOP supports combining multiple traces into one program (§3.5). To evaluate this feature, we use KLEE [12] to generate test cases as inputs for program tracing by TOP. To simplify the experiment, we use the same KLEE command for coreutils programs[2]. The 7^{th} and 8^{th} columns of Table 2 show the number of test cases and the coverage for each program (use `klee-stats` command), respectively. The final source code size of these programs is reported in the 9^{th} column of Table 2. Meanwhile, the average offline combination time (not shown in Table 2) is 0.18 second.

4.2 Evaluation with Obfuscated Windows Binaries

Obfuscated binaries, especially those of malware, abound on Windows. The goal of binary code obfuscation is to disrupt analysis of the code and deter reverse engineering efforts. In general, there are three types of widely used binary analysis platforms: disassembler, debugger, and virtual machine (VM). Consequently, obfuscation techniques can be categorized into anti-disassembler, anti-debugger, and anti-VM. For each category, there exist a variety of techniques. For example in the anti-dissembler category, there exist the techniques of garbage code insertion, control flow obfuscation, instruction aliasing, binary code compression, and encryption.

To evaluate the resilience of TOP against these obfuscation techniques, we select 15 representative obfuscated samples from [5] (shown in Table 3), which cover the state-of-the-art obfuscation techniques. The samples and their source code, plus two additional binary-only, packed samples, are from *offensivecomputing.net*. Note that a reason for selecting the 15 samples is that they allow us to verify the recovered code's correctness. For the two packed malware samples, we have no knowledge about their implementation, except that they are packed.

Effectiveness. Interestingly, many of these binary samples have only 1536 bytes as shown in the 2^{nd} column of Table 3. The reason is that their source code is very small, each containing only a few lines of code for simple demonstration of an obfuscation technique. That also explains why the recovered source code is also small (shown in the 9^{th} column). For all the obfuscated samples, TOP successfully recovers their source code from traces. We have run the recovered and recompiled programs and verified that they have consistent semantics with the original binaries (last column).

To better illustrate the strength of TOP, consider the two pieces of assembly code in Table 4, one from the sample `middle_instruction.exe` for instruction aliasing, and the other from `garbage_bytes.exe` for adding garbage code. For instruction aliasing, we see that the original code's execution at line 3 will jump to the middle of the instruction at line 1 as the condition for `jz` is always true (by `xor` in line 2). Then `eb 05` in line 1 would be disassembled as a direct `jmp` which goes to line 5. In other words, the garbage byte in line 4 will never be executed. However, IDA Pro cannot disassemble this obfuscated code because it would disassemble `db` as opcode `call` at line 5 in the second column. Due to incorrect alignment of instructions, the subsequent disassembling would be incorrect. In contrast, TOP dynamically generates the correct results as shown in the third column. Note that in line 6, TOP adds a safety guard (`jmp`) to handle partial coverage issue. For the second example, since a garbage byte `0x6a` in line 3 is introduced, IDA Pro fails to perform static disassembling. In both cases, we can compile the code generated by TOP.

Performance overhead. With the small size of the majority of these samples, TOP quickly recovers their source code within several milliseconds. For `Win32.Bamital.exe` and `Virus.Win32.Adson.exe` which are real-world malware, it takes TOP about 14 seconds to perform online tracing. The reason for the much

[2] http://ccadar.github.io/klee/CoreutilsExperiments.html

Original Assembly Code	Disassembly from IDA Pro	Disassembly from TOP

`66 b8 eb 05    mov    ax,0x05eb`
`31 c0          xor    eax, eax`
`74 fa          jz     $-4`
`e8             db 0xe8 ;garbage byte`
`58             pop    eax`
`(middle_instruction.exe)`
	`loc_401006:`
	`mov    ax, 5EBh`
	`xor    eax, eax`
	`jz     loc_401008`
	`call   near ptr 6A98686Bh`
		`mov    ax, 0x5eb`
		`xor    eax, eax`
		`jz     loc_0x401008`
		`jmp    loc_ERROR`
		`loc_0x401008:`
		`jmp    loc_0x40100f`
		`loc_0x40100f:`
		`pop    eax`

`31 c0          xor    eax, eax`
`74 01          jz     .destination`
`6a             db 0x6a ;garbage byte`
`               .destination:`
`58             pop    eax`
`(garbage_bytes.exe)`
	`xor    eax, eax`
	`jz     loc_401007+1`
	`loc_401007:`
	`push   58h`
		`xor    eax, eax`
		`jz     loc_0x401008`
		`jmp    loc_ERROR`
		`loc_0x401008:`
		`pop    eax`

Table 4: Disassembling results from IDA Pro and TOP for obfuscated programs.

longer time is that both malware binaries involve many iterations for decryption and decompression.

Coverage. Unlike the Linux samples, we do not have KLEE to generate test cases for the obfuscated binaries to improve coverage. Instead, we perform a manual check on the coverage for the obfuscated samples with source code, as reported in the 10^{nd} column. The TOP-generated code has high coverage, with 5 of them having 100% coverage. For the two malware samples, we do not estimate their coverage due to lack of ground truth.

5. APPLICATIONS

With the capability of translating binary execution traces into C code, TOP enables a variety of security applications. In this section, we demonstrate two such applications: malware unpacking and identification, and security function transplanting.

5.1 Malware Unpacking and Identification

Many malware programs today are heavily armored with anti-analysis mechanisms to make analysis of them difficult. Binary code packing is the most common anti-reverse engineering technique. According to a recent report, 34.79% of the malwares is packed [49]. An earlier research paper [35] reports that over 80% of malware is packed. Dynamic analysis has been shown a promising approach to unpacking malware [50, 42, 36, 35, 54].

Unpacking. Based on dynamic analysis, TOP naturally possesses the capability of unpacking malware. To this end, we design an unpacking plugin based on TOP. As we trace the entire execution of a binary (using layers), we are able to detect the unpacking code and the real program code. The real program code is the one that is finally settled in the memory and executed. It might happen that a packer could alternatingly execute the unpacking code and the real code (though we have not seen such a case in the wild). In that case, we might not be able to identify the real code based on full lifetime tracing. Still TOP will output all the traced code, which can be further analyzed to identify the unpacking code.

To test TOP's unpacking capability, we use 10 publicly available packers often used by malware authors and by researchers of many related efforts [50, 42, 36, 35, 54]). For the "testing goat" program, we use Windows `tasklist.exe`, a command console program with a binary of 77824 bytes, to generate packed samples. The samples are created using the default configuration for all the packers. To test multi-layer packing, we deliberately create two samples that are packed by two packers. Finally we have 12 samples: 10 are packed once and two are packed twice. We run all these samples without any command line option.

The evaluation results are reported in Table 5. The first column shows the size of packed `tasklist.exe`, and the second column shows the packer used for packing. We report the numbers of symbolized addresses, safety checks, traced functions, and lines of code, of the generated source code. The TOP unpacking plugin can successfully detect the unpacking routine and the real `tasklist.exe` code, even if the sample binary is packed by two packers. Interestingly, as shown in the last 4 columns of Table 5, the real `tasklist.exe` programs generated by tracing the 12 samples (under the same configuration) have identical results across the samples. Next, we recompile the generated source code of `tasklist.exe` and execute it with the same command line option of the original program. We confirm that the outputs are the same as that generated by the original `tasklist.exe`.

Identification. Having been able to unpack malware, TOP can be further applied to malware identification. As shown in Table 5, each packer has its own distinctive features; and the code that follows the unpacking code belongs to the real program code. As such, we can identify the prefix of the recovered code to identify packers, and eliminate the unpacking code to expose the real program code. In particular, we could perform source code diff-ing (text-based) to identify the unpacking prefix; or we could build and normalize the program control structures (e.g., CFGs) to identify the prefix.

To perform malware identification, we first generate a suite of (un)packer signatures. Given an unknown binary, we run it using TOP. By comparing the output with the signatures, we eliminate the unpacking code. Then, we can apply various techniques to identify/classify the real malware code (e.g., source code diff-ing and control structure comparison). In our experiment, for simplicity, we use source text diff-ing, which indicates that the recovered programs from the samples are actually the same piece of malware. This is also confirmed by the last four columns of Table 5.

5.2 Security Function Transplanting

Binary code reuse is meaningful for both goodware and malware. To extract the binary code of interest, a user only needs to designate the entry point and exit point, and TOP will automatically translate the executed instructions into C code. This code includes all the functions called and the symbolized global data accessed.

Goodware Function Reuse. We take an MD5 hash algorithm implementation as an example. Program `md5sum` (from coreutils) is a widely used cryptographic hash tool whose implementation contains an important function `digest_file` (instruction address: 0x8049f70 in our experiment) which computes the MD5 digest of a given file.

Binary Size (KB)	Packer	Unpacking Code				Recovered `tasklist.exe` Code			
		#Fun	#LOC	#Symbolized	#Safety	#Fun	#LOC	#Symbolized	#Safety
34.00	UPX	1	169	34	1	144	5748	1171	271
39.50	ASPack	14	913	140	16	144	5748	1171	271
363.50	ASProtect	19	504	86	3	144	5748	1171	271
46.46	RlPack	30	734	145	22	144	5748	1171	271
34.00	MPress	9	941	126	20	144	5748	1171	271
32.16	Mew	11	616	114	0	144	5748	1171	271
34.04	XComp	9	282	84	2	144	5748	1171	271
33.66	XPack	9	262	80	2	144	5748	1171	271
29.98	WinUnPakc	17	390	49	2	144	5748	1171	271
35.50	PEcompact	6	171	31	0	144	5748	1171	271
42.50	ASPack	14	864	130	16	144	5748	1771	271
	ASPack	14	913	140	16				
39.50	ASPack	14	797	113	16	144	5748	1771	271
	XPack	9	262	80	2				

Table 5: Evaluation results with 10 publicly available packers using `tasklist.exe` as a testing goat program.

To avoid the effort of re-implementation, a programmer can run `md5sum` binary on TOP with tracing/translation entry point being 0x8049f70 and exit point being 0x804a07e (the `ret` instruction of function `digest_file`). TOP automatically generates the source code of all the executed functions during tracing, including `fun_0x8049f70` and 11 other sub-routines, symbolizes 151 addresses, and adds 32 checks. Next, the generated function can be reused as a normal C function in developing new software. We have successfully developed a file comparison program that performs MD5 hash check for two input files by reusing function `fun_0x8049f70` and its subroutines.

Malware Function Reuse. For demonstration purpose, we target three functions – two for environment detection and one for stream cipher – in malware code and show how to reuse them. The environment detection functions are from the two samples `anti-vm_in_instruction.exe` and `hardware_bp.exe` we have tested earlier. Our purpose is to extract the functions for VM and debugger detection and reuse them in new programs. We have successfully extracted these two functions and linked them with other software components.

We have also extracted an `RC4` stream cipher function from `Worm: Win32/Sality.AU`. Since the malware is packed, we run TOP to get the trace of the whole program, from which we identify the `RC4` cipher function. Our manual analysis reveals that the entry address of the `RC4` function is 0x401212 and the exit address is 0x402723. We then run the malware on TOP again, symbolizing 86 memory addresses and adding 24 checks for this function and its 5 callees. Now we can write a new program to reuse the cipher function and confirm the correctness of its functionality.

6. DISCUSSIONS AND FUTURE WORK

The main limitation of TOP is the incomplete coverage that arises from its dynamic analysis approach. The recovered code only reflects the traced behavior and rejects the behavior that is not exercised. As demonstrated in our evaluation, path coverage can be improved by advanced program testing techniques (e.g., symbolic execution [12, 14]). Part of our future work is to better integrate these techniques into TOP.

The current focus of TOP is its source code reconstruction and reuse capability. Hence we have not attempted to make the recovered code more readable or optimized. Our future work will address the readability issue by introducing a richer set of structures (e.g., `do-while` and `for` structures for loops) and leveraging advanced decompilation techniques (e.g., Hex-Rays [2], Boomerage [27], and Phoenix [51], which tend to achieve better readability). We also plan to optimize the recovered code (e.g., by eliminating unnecessary safety guards). Consider the `middle_instruction.exe` example in Table 4. We could have removed the `jmp loc_ERROR` guard because `xor eax, eax` will clear the zero flag.

TOP is currently platform and environment dependent. More particularly, it requires the same kernel and library support to compile and execute the reconstructed code. It also relies on the presence of the same needed external resources as the original executions, such as configuration files.

TOP currently has an effective scheme to ensure control flow safety, which is the challenge we have encountered in the programs we consider. It is possible that more complicated and subtle safety conditions may arise when TOP is applied to more complex programs. Also, the offline trace merging process simply merges control flow paths, which may not be sufficient when more extremal situations are encountered (e.g., when executions are non-deterministic). We plan to further investigate these issues.

TOP may not be able to recover the source code of all binaries. For example, a malware program may involve virtualization-based obfuscation [53]. Even though TOP can generate its source code from traces, the virtualization code will be recovered as well. Moreover, TOP cannot handle hypervisor-level malware such as the red pill, due to the lack of hypervisor-level tracing capability. This limitation will be addressed by our future work.

It is well known that broken dependences caused by control flows pose an issue for taint analysis in general. However, that is *not* an issue for TOP because we use taint analysis in a very restricted context. Specifically, to handle indirect control transfers, we use it to back track from an indirect invocation to the original instruction that loads the indirect target and replace the target with a symbol to ensure that the corresponding function becomes relocatable. The propagation from the original load of the target and the invocation must be via data dependence. It is possible that the invocation is relevant to other instructions through control dependences, yet it is unnecessary to symbolize those relevant instructions. While it is possible that aggressive obfuscation may cause problems for TOP in the future, we have not encountered such a case in our experiments.

7. RELATED WORK

Decompilation. Decompilation is the process of reconstructing program source code from code in lower-level languages (e.g., assembly or machine code) [16, 6]. Tools like HexRay [2], Boomerage [27], and Phoenix [51] offer a variety of techniques to elevate low-level assembly instructions to higher-level source code. While these techniques are all based on static analysis, TOP is based on dynamic analysis, one of the first decompilation techniques to do so.

Binary Code Extraction and Reuse. Our work is closely related to BCR [9], Inspector Gadget [38], Virtuoso [24], and VMST [29]. While the discussion of these systems can be found in §1, we note again that TOP is a novel technique that translates all executed instructions into C code, with automatic addition of safety checking code. Moreover, the recovered code is readily reusable and incurs negligible performance overhead compared with the original binary (§4). Moreover, TOP does not focus on identification of functional components for reuse.

Most recently, in a parallel effort of TOP we proposed Bistro [23], which is a binary manipulation, rewriting and instrumentation infrastructure that allows *binary-to-binary* functional component extraction (from one binary) and implanting (in another binary). Different from TOP, Bistro operates directly and statically on binaries.

Dynamic Data Dependency Tracking. TOP leverages dynamic data dependence tracking (i.e., taint analysis) to symbolize memory addresses. Data dependence tracking has been widely applied to many security applications, such as data lifetime tracking [15], exploit detection [20], vulnerability discovery [44, 28, 13, 34, 32], protocol and data structure reverse engineering (e.g., [11, 21, 56, 10, 40, 55], and malware analysis [25]). TOP does not make any new advances in taint analysis per se but demonstrates its new application – memory address symbolization.

Malware Analysis and Unpacking. Unpacking aims at uncovering the original malicious code which had been packed by a variety of (binary) transformations. Unpacking techniques exist, such as PolyUnpack [50], OmniUnpack [42], Renovo [36], Justin [35], and Eureka [54]. There also exist a wide range of malware analysis techniques [45, 26, 37, 38]. TOP complements these efforts and enables useful malware analysis capabilities such as malware identification and function extraction.

8. CONCLUSION

We have presented trace-oriented programming (TOP), a new framework to enable the reuse of legacy binary code from execution traces. Through dynamic execution of a binary, TOP collects necessary information such as control structures, memory addresses and accesses, and safety information; and then translates each executed instruction into a predefined template or inlined assembly according to its semantics. While TOP shares the same goal with existing decompilation techniques, it enjoys unique benefits from dynamic analysis, such as being obfuscation resilient and free from point-to analysis. We have implemented a proof-of-concept TOP prototype. Our evaluation results with over 100 legacy binaries (including malware binaries) indicate the effectiveness, efficiency, and safety of TOP and demonstrate the application of TOP to malware analysis and security function reuse.

Acknowledgement

We thank the anonymous reviewers for their insightful comments. Kenneth A. Miller is supported by a DoD scholorship under contract H98230-12-1-0452. This research is funded by DARPA under contract 12011593. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the funding agencies.

9. REFERENCES

[1] Intel-64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C.

[2] Hex-rays decompiler SDK. http://www.hex-rays.com/.

[3] Making a disassembler: Instruction aliasing. http://trusted-disassembler.blogspot.com/2012/12/instruction-aliasing.html.

[4] QEMU: an open source processor emulator. http://www.qemu.org/.

[5] BRANCO, R. R. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. In *Black Hat Technical Security Conf.* (Las Vegas, Nevada, July 2012).

[6] BREUER, P. T., AND BOWEN, J. P. Decompilation: The enumeration of types and grammars. *ACM Trans. Program. Lang. Syst. 16*, 5 (1994), 1613–1647.

[7] BRUMLEY, D., AND NEWSOME, J. Alias analysis for assembly. Tech. Rep. CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006.

[8] BURKE, M. G., CARINI, P. R., CHOI, J.-D., AND HIND, M. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, 1995), Springer-Verlag, pp. 234–250.

[9] CABALLERO, J., JOHNSON, N. M., MCCAMANT, S., AND SONG, D. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)* (San Diego, CA, February 2010).

[10] CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and and Communications Security (CCS'09)* (Chicago, Illinois, USA, 2009), pp. 621–634.

[11] CABALLERO, J., AND SONG, D. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS'07)* (Alexandria, Virginia, USA, 2007), pp. 317–329.

[12] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (San Diego, CA, 2008).

[13] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)* (Alexandria, Virginia, USA, 2006), ACM, pp. 322–335.

[14] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (Newport Beach, California, USA, 2011), ASPLOS '11, pp. 265–278.

[15] CHOW, J., PFAFF, B., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding data lifetime via whole-system simulation. In *Proceedings of the 13th USENIX Security Symposium* (2004).

[16] CIFUENTES, C. Reverse Compilation Techniques. *PhD thesis, Queensland University of Technology* (1994).

[17] CIFUENTES, C., AND GOUGH, K. J. Decompilation of binary programs. *Softw. Pract. Exper. 25*, 7 (July 1995), 811–829.

[18] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. *Technical Report 148, Department of Computer Science, University of Auckland* (1997).

[19] CONSTANTIN, L. Decompiled stuxnet code published online, 2011. http://news.softpedia.com/news/Anonymous-Publishes-Decompiled-Stuxnet-Code-184448.shtml.

[20] CRANDALL, J. R., WU, S. F., AND CHONG, F. T. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim. 3*, 4 (2006), 359–389.

[21] CUI, W., PEINADO, M., CHEN, K., WANG, H. J., AND IRUN-BRIZ, L. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)* (Alexandria, Virginia, USA, October 2008), pp. 391–402.

[22] Debray, S. K., Muth, R., and Weippert, M. Alias analysis of executable code. In *Symposium on Principles of Programming Languages (POPL'98)* (1998), pp. 12–24.

[23] Deng, Z., Zhang, X., and Xu, D. Bistro: Binary component extraction and embedding for software security applications. In *Proceedings of 18th European Symposium on Research in Computer Security (ESORICS'13)* (Egham, UK, September 2013), LNCS.

[24] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the* 32^{nd} *IEEE Symposium on Security and Privacy* (Oakland, CA, USA, 2011), pp. 297–312.

[25] Egele, M., Kruegel, C., Kirda, E., Yin, H., , and Song, D. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference (Usenix'07)* (June 2007).

[26] Egele, M., Kruegel, C., Kirda, E., Yin, H., and Song, D. Dynamic spyware analysis. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA, 2007), USENIX Association, pp. 1–14.

[27] Emmerik, M. V., and Waddington, T. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering* (2004), pp. 27–36.

[28] Forrester, J. E., and Miller, B. P. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium* (Seattle, Washington, 2000), USENIX Association, pp. 1–10.

[29] Fu, Y., and Lin, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of* 33^{rd} *IEEE Symposium on Security and Privacy* (May 2012).

[30] Fu, Y., and Lin, Z. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments* (Houston, TX, March 2013).

[31] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'03)* (February 2003).

[32] Godefroid, P., Kiezun, A., and Levin, M. Y. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)* (Tucson, AZ, USA, 2008), ACM, pp. 206–215.

[33] Godefroid, P., Klarlund, N., and Sen, K. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)* (Chicago, IL, USA, 2005), ACM, pp. 213–223.

[34] Godefroid, P., Levin, M., and Molnar, D. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (San Diego, CA, February 2008).

[35] Guo, F., Ferrie, P., and cker Chiueh, T. A study of the packer problem and its solutions. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)* (Boston, USA, September 2008).

[36] 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* (Alexandria, Virginia, USA, 2007), ACM, pp. 46–53.

[37] 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 conference on USENIX security symposium* (Montreal, Canada, 2009), pp. 351–366.

[38] Kolbitsch, C., Holz, T., Kruegel, C., and Kirda, E. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of 2010 IEEE Security and Privacy* (Oakland, CA, May 2010).

[39] Liang, D., and Harrold, M. J. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)* (Toulouse, France, 1999), Springer-Verlag, pp. 199–215.

[40] Lin, Z., Zhang, X., and Xu, D. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)* (San Diego, CA, February 2010).

[41] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)* (Chicago, IL, USA, 2005), pp. 190–200.

[42] Martignoni, L., Christodorescu, M., and Jha, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)* (2007), pp. 431–441.

[43] Miecznikowski, J., and Hendren, L. J. Decompiling java bytecode: Problems, traps and pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction* (London, UK, UK, 2002), CC '02, Springer-Verlag, pp. 111–127.

[44] Miller, B. P., Fredriksen, L., and So, B. An empirical study of the reliability of UNIX utilities. In *Proceedings of the Workshop of Parallel and Distributed Debugging* (1990), Academic Medicine, pp. 9–19,.

[45] Moser, A., Kruegel, C., and Kirda, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 231–245.

[46] Mycroft, A. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems (ESOP'99)* (London, UK, 1999), Springer-Verlag, pp. 208–223.

[47] Newsome, J., and Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)* (San Diego, CA, February 2005).

[48] Pearce, D. J., Kelly, P. H., and Hankin, C. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst. 30*, 1 (2007), 4.

[49] Rodrigo Rubira Branco, G. N. B., and Neto, P. D. Scientific but not academical overview of malware anti-debugging, anti-disassembly and antivm technologies. Tech. rep., "NOSPAM" qualys.com, Qualys-Vulnerability and Malware Research Labs.

[50] Royal, P., Halpin, M., Dagon, D., Edmonds, R., and Lee, W. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 289–300.

[51] Schwartz, E. J., Lee, J., Woo, M., and Brumley, D. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22nd USENIX Security Symposium* (Washington DC, USA, 2013), USENIX Association.

[52] Sen, K., Marinov, D., and Agha, G. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)* (Lisbon, Portugal, 2005), ACM, pp. 263–272.

[53] Sharif, M., Lanzi, A., Giffin, J., and Lee, W. Automatic reverse engineering of malware emulators. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* (2009), SP '09, pp. 94–109.

[54] Sharif, M., Yegneswaran, V., Saidi, H., and Porras, P. Eureka: A framework for enabling static analysis on malware. In *Proceedings of the 13th European Symposium on Research in Computer Security* (Malaga, Spain, October 2008), LNCS.

[55] Slowinska, A., Stancescu, T., and Bos, H. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)* (San Diego, CA, February 2011).

[56] Wondracek, G., Milani, P., Kruegel, C., and Kirda, E. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (San Diego, CA, February 2008).