Binary Code Is Not Easy

Xiaozhu Meng and Barton P. Miller
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706, USA
{xmeng,bart}@cs.wisc.edu

ABSTRACT

Binary code analysis is an enabling technique for many applications. Modern compilers and run-time libraries have introduced significant complexities to binary code, which negatively affect the capabilities of binary analysis tool kits to analyze binary code, and may cause tools to report inaccurate information about binary code. Analysts may hence be confused and applications based on these tool kits may have degrading quality. We examine the problem of constructing control flow graphs from binary code and labeling the graphs with accurate function boundary annotations. We identified several challenging code constructs that represent hard-to-analyze aspects of binary code, and show code examples for each code construct. As part of this discussion, we present new code parsing algorithms in our open source Dyninst tool kit that support these constructs, including a new model for describing jump tables that improves our ability to precisely determine the control flow targets, a new interprocedural analysis to determine when a function is non-returning, and techniques for handling tail calls. We evaluated how various tool kits fare when handling these code constructs with real software as well as test binaries patterned after each challenging code construct we found in real software.

CCS Concepts

•**Software and its engineering** → **Automated static analysis;** *Software reverse engineering;*

Keywords

Static binary code analysis; Challenging code constructs; Jump table model

1. INTRODUCTION

Binary code analysis is used in a wide range of applications, including performance analysis [1, 15, 33], software reverse engineering [12, 18], debugging [2], software reliability [31], software forensics [42] and security [19, 23, 36]. The analysis of binary code is a critical capability in these applications because it does not require source code to be available and targets the actual software artifact that is executed. Even when you have the source code, experience has shown that the semantics of the binary code that is executed can be different from the source code [4].

Binary code analysis can be static or dynamic. In this paper, we focus on static analysis as it is a foundational technique in many areas including dynamic analysis (such as for analyzing self-modifying code and packed malware [9, 38, 44, 51]). It has the advantages that it does not require a program to be executed and its analysis coverage does not depend on the coverage of the available input sets.

Previous studies on static binary code analysis have focused on identifying and addressing challenging code constructs in binary code, including identifying function entry points [5, 21, 43], resolving indirect control flow [6, 13, 16, 26, 40, 47], and disambiguating non-code bytes [46]. The goal of this paper is to improve the handling of these three constructs and expand our study to include additional challenging code constructs to explore complexities that have been introduced into binary code by modern compilers and run-time libraries.

These code complexities influence the ability of an analyst to understand the operation and intent of a program, and the ability of a tool to correctly instrument or transform the binary program to trace, debug, test, monitor, or sandbox it. Supporting these code constructs in our own open source Dyninst tool kit [37] brings a universal benefit, as Dyninst is widely used in building debugging tools including STAT [2] and SystemTap [17], performance tools including COBI [34], Extrae [30], HPCToolKit [1], and Open|SpeedShop [45], and many other tools for security analysis [19, 23, 39, 44, 50, 53] and reverse engineering [10, 24, 27, 41, 43].

Binary code analysis tool kits [3, 11, 22, 37, 47] provide several capabilities to help users automate the process of binary code analysis. The capabilities include decoding bytes into machine instructions, understanding the instruction semantics, performing control flow and dataflow analyses, and assigning source language semantics to binary code. Each of these capabilities can build on and interact with the previous ones; the last one, the assigning of source code semantics to the binary code is subtle because there can be more than one reasonable and consistent assignment.

Decoding bytes into machine instructions is the first step of binary code analysis. This capability is straightforward when you know the start address of an instruction. However, there are many cases where the starting address of an instruction is not obvious. It can be difficult to find the starting address

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany

http://dx.doi.org/10.1145/2931037.2931047

of functions in the case where you have few, if any symbols in the executable file ("stripped" code). This lack of symbols is common in both malicious code and production releases of conventional code. Even if you can find the start of a function, indirect control flow within the function can make it difficult to find all the code. In practice, code analysis tool kits struggle with this issue and often miss real instructions or report bogus instructions. Common problems include reporting padding bytes inserted by compiler as real instructions, missing instructions that share bytes (which, surprisingly, occurs not only in malware but in conventional code), interpreting data bytes as code, and misinterpreting code as data bytes.

Building a control flow graph (CFG) from binary code describes the basic structure of the program. It also lays foundation for dataflow analyses and robust binary instrumentation and modification [7, 8]. However, tool kits often produce inaccurate CFGs failing to recognize non-returning functions and imprecisely handling indirect control flow.

Assigning source language semantics to binary code is a more interesting problem, which represents binary analysis results with familiar constructs such as functions, loops, function arguments, and local variables. Such functionality is necessary for the programmer to understand the program in terms with which they are familiar, to provide a reasonable labeling when the programmer has the source code, and to provide concrete targets for program instrumentation and modification. However, tool kits often have a difficult time identifying these constructs. Common problems include not understanding that functions are no longer contiguously allocated in memory, functions can interleave, and functions can share code. These oversimplifications can cause inaccurate correspondence between binary code and source code.

Tools built on top of binary code analysis suffer when analysis tool kits provide inaccurate information. If a performance analysis tool is provided an inaccurate correspondence between the binary and source code, profiling data may be attributed to wrong locations in source code, causing users to miss-identify performance bottleneck [1]. Security applications need accurate information about the binary to avoid missing attacks or reporting false alarms [23, 52]. In addition, dataflow analyses can be imprecise if the CFG is not accurate. Binary instrumentation [8, 32] often uses register liveness analysis to figure out which registers can be used by instrumentation without introducing spills; accurate dataflow analysis is essential here. Tools built on top of binary analysis tool kits almost always assume that underlying tool kits provide accurate information and can misbehave if the information in not accurate [1, 23, 31].

In this paper, we examine the problem of constructing CFGs from binary code and labeling the CFG with accurate function boundary annotations. Addressing this problem requires the interacting capabilities of finding instructions, building the CFG, and assigning source function semantics. We split the problem into three analysis stages:

- *code discovery*, finding all instructions in the binary;
- *CFG construction*, determining the basic blocks and connecting the edges between them (and knowing when *not* to connect the edges); and
- *CFG partitioning*, labeling edges as inter-procedural or intra-procedural to determine the function boundaries.

From our experience in building a binary analysis tool kit, we have identified eight challenging code constructs found in real code that often confuse tools. For these constructs, we use code examples to discuss why they are difficult and present our strategies for handling them. In particular, we present a new model describing jump tables that improves our ability to precisely determine the control flow targets, a new interprocedural analysis for determining whether a function returns, and techniques for handling tail calls, overlapping functions, and overlapping instruction sequences.

We used SPECint 2006 and created test binaries that are patterned after each challenging code construct to evaluate several commonly-used binary analysis tool kits, including BAP [11], GNU Objdump [20], IDA Pro [22], Jakstab [25], OllyDbg [35], SecondWrite [47], and Dyninst [37]. Our results show that these challenging code constructs are prevalent in real software and most of these tool kits can be confused by challenging code constructs, so are likely to provide inaccurate information about the binary in these cases. The underlying message of such a study is that while building a binary analysis tool kit for common code constructs is a well-understood task, handling the full spectrum of code generated by a modern compiler adds significant work.

We present basic definitions as background in Section 2. In Section 3, we overview our eight challenging code constructs and discuss them in detail from Section 4 to Section 6. We present our evaluation comparing existing binary analysis tool kits in Section 7 and conclude in Section 8.

2. BASIC DEFINITIONS

The problem of constructing and labeling the CFG can be stated as: given a *program*, we extract a *CFG* and a set of *functions*. Previous efforts on this problem have made various simplifying assumptions on definitions of a *program*, *CFG*, and *function* [4, 5, 13, 26, 40, 48]. Common assumptions include that the program contains relocation information [46], function calls always return [29], and function bodies are independent and laid out contiguously in memory [5, 26]. However, the simplified definitions do not always hold true with real world binaries and are not sufficient to represent the complexities of real world binaries. We present definitions of *program*, *CFG*, and *function* from Bernat and Miller [8]. These definitions do not impose unnecessary assumptions on the binary; thus they are suitable to represent the challenging code constructs that we discuss in this paper.

Definition 1 (Program) A program P is defined as a tuple $P = (C, D)$, where $C =< i_0, i_1, \ldots, i_m >$ is a sequence of instructions that P may execute and D represents data.

Bernat and Miller point out that this definition is sufficiently permissive to present real world binaries: it does not assume the existence of symbol, debugging or relocation information; C and D can interleave in memory; instructions in C can overlap.

Definition 2 (CFG) A CFG is defined to be a directed graph $G = (V, E, V_e, V_x, T)$, where

$V = B \cup \{v_\perp\}$ is a set of nodes corresponding to all basic blocks B and a special *sink* node $v_\perp$ that has no instructions or outgoing edges;

$E \subseteq V \times V$ is a set of control flow edges between nodes;

$V_c \subseteq V$ is a set of entry nodes;

$V_x \subseteq V$ is a set of exit nodes;

$T : E \rightarrow$ {intraprocedural, interprocedural} assigns a label to an edge.

The basic blocks B are defined in a conventional way. Each basic block $b =< i_0, i_1, \ldots, i_n >$ is a consecutive instruction sequence with i_0 being the only entry and i_n being the only exit. The sink node $v_\perp$ is used to represent unknown control flows [8, 11, 26, 48], mainly caused by indirect jumps and indirect calls.

Definition 3 (Function) Let F be the set of functions in the program. A function is a subgraph of the CFG $f_i = (v_i, V_i, E_i, X_i)$, where

$v_i \in V$ is the entry node of the function;

$V_i \subseteq V$ is the set of nodes of the functions; $V_i = \{v \in V \mid v$ is reachable from v_i by traversing only intraprocedural edges$\}$

$E_i \subseteq E$ is the set of intraprocedural edges between V_i;

$X_i \subseteq V_i$ are the exits nodes of the function.

Under this definition for a function, functions can share code, be interleaved, and be non-contiguous in memory. We can also represent a function that has multiple entry points as several single-entry-point functions sharing code. Some previous projects have defined a function as an interval of addresses [22, 26, 35]. Their definition cannot correctly model these challenging code constructs.

While the above definitions are applicable to any ISA, in this paper, we focus on x86 and x86-64 as they are commonly used platforms. Their instructions have variable lengths, making it more challenging to distinguish data from code and identify padding bytes. We focus on stripped binaries, as have several previous projects [4, 5, 21, 43]. Parsing stripped binaries is significantly more challenging than parsing binaries with symbols and debugging information. However, we need to be able to handle stripped binaries because binary code is often stripped in real world. Being able to handle stripped binaries also provides a foundation to analyze malicious code.

3. CODE CONSTRUCTS OVERVIEW

From our experience in building a binary analysis tool kit, we identified eight challenging code constructs. These code constructs have often confused existing binary analysis tool kits. Tool kits may miss real instructions, report bogus control flows or inaccurately label function boundaries. The above inaccuracies in binary analysis are critical to recognize as they may prevent analysts from understanding the structure and intent of a program and cause binary instrumentation and modification to be unsafe, incorrect, or incomplete. In this section, we present an overview of the challenging code constructs, as summarized in Table 1. The code constructs are classified into the following three analysis stages: *code discovery*, finding all instructions in C that a program may execute (Section 4); *CFG construction*, building nodes and edges in G (Section 5); and *CFG partitioning*, determining which parts of the CFG belong to which functions (Section 6). At the end of this section, we discuss the relations between the three analysis stages.

3.1 Code Discovery

We identified three code constructs that make code discovery difficult. If the three constructs are handled improperly, binary analysis tool kits may misinterpret critical data bytes as instructions or miss real instructions. Binary instrumentation and modification may then modify critical data bytes and cause programs to crash. Instrumentation and modification may also be incomplete because real instructions are missed, which may not be tolerable in security applications. The three challenging code constructs are:

Non-code bytes: code must be distinguished from non-code bytes that appear in code sections, such as jump tables, read-only data and padding bytes. The compiler may insert padding bytes between instructions to align instructions and increase cache efficiency. It is not trivial to distinguish these non-code bytes from real code because the non-code bytes can usually be decoded into valid instructions. Note that even though the compiler may put read-only data and jump tables into separate read-only data sections, this is not required. In fact, we find that Windows system libraries usually do not contain a read-only data section; read-only data and jump tables are embedded in code sections.

Missing symbols: the symbol table of a program is incomplete, missing, or inaccurate. Binary analysis tool kits often use function symbols to identify function entry points. Without complete and accurate symbols, this task becomes significantly more difficult.

Overlapping instructions: multiple instructions share bytes. This code construct is only present on architectures that instructions have variable lengths and the start address of an instruction is not required to align, such as the x86 and x86-64. If binary analysis tool kits assume that instructions never share bytes, they will miss real instructions.

3.2 CFG Construction

We identified two challenging code constructs for CFG construction. Handling them inappropriately may cause binary analysis tool kits to miss real control flow and report bogus control flow. The inaccuracy in a CFG can confuse analysts and degrade the quality of tools that are based on binary analysis. For example, structured binary editing marks functions unmodifiable if the functions contain unresolved intraprocedural indirect control flow [8]. The two code constructs are:

Indirect control flow: this code construct refers to indirect jump instructions and indirect call instructions. Indirect control flow is mainly used to implement pointer-based control flow, virtual functions and switch statements. The control flow targets are dynamically calculated and it is hard to accurately determine them statically. In this paper, we focus on jump tables, which are a set of indirect control flow where the calculations of the control flow targets are based on a well understood structure. Jump tables often represent intraprocedural control flows and it is essential to resolve them precisely for code discovery and applications such as structured binary editing [8].

Non-returning functions: a function call to a non-returning function will never return to this call site. Often the compiler knows whether a call will return or not, so will safely put unrelated code from the same function or code from another function immediately after a non-returning call. If a binary analysis tool kit cannot recognize non-returning functions, it will wrongly report that control flow continues from a non-returning call to its next block.

3.3 CFG Partitioning

We identified three code constructs in CFG partitioning. Not being able to handle them may cause binary analysis tool kits to inaccurately label function boundaries, which can cause problems in binary instrumentation and modification. Two common instrumentation operations are instrumenting the entries of all basic blocks of a given function and instrumenting function entries and exits. If the function boundaries are inaccurate, we may instrument at wrong places or miss

Table 1: An overview of identified challenging code constructs

Stage	Code construct	Challenge	Discussion
Code discovery	Non-code bytes	Distinguish whether a byte in code sections is code or not	Section 4.1
	Missing symbols	Identify function entry points	Section 4.2
	Overlapping instructions	Identify all instructions that share bytes	Section 4.3
CFG construction	Indirect control flow	Precisely determine the targets of an indirect control flow instruction, with an emphasis on jump tables	Section 5.1
	Non-returning functions	Identify all non-returning functions. A function call to such a function should not have an control flow edge to the next basic block.	Section 5.2
CFG partitioning	Functions sharing code	Correctly represent the shared blocks of code in all functions that share them	Section 6.1
	Non-contiguous functions	Correctly represent a non-contiguous function where other functions' code may be mixed in between	Section 6.1
	Tail calls	Distinguish whether a jump instruction is targeting the entry point of another function or targeting an address inside the same function	Section 6.2

Initial state: `%ebx`=0x80d6378 and 0 ≤ `%eax` ≤ 12

Address	Instruction	Byte	Table value	
80b15e5	`mov %ebx,%ecx`			Jump table code
80b15e7	`sub -0x24d88(%ebx ,%eax,4),%ecx`			
80b15ee	`jmp *%ecx`			
80b15f0 80b15f1	`loopne 80b163e`	e0 4c	0x24ce0	Jump table address list
80b15f2 80b15f3	`add (%eax),%al`	02 00		
...				
80b1620	`pop %esp`	5c	0x24a5c	
80b1621	`dec %edx`	4a		
80b1622 80b1623	`add (%eax),%al`	02 00		

Figure 1: An example of non-code bytes embedded in code sections from libc. The code in the address range [80b15e5, 80b15ef] is a jump table calculation that uses the table at address range [80b15f0, 80b1623]. We get valid Pentium instructions if the jump table bytes in gray are decoded as code.

program places where we should instrument. The three code constructs are:

Functions sharing code: functions can share blocks of code. Functions may have common functionality, which leads to share the same blocks of code, like error handling code and stack tear-down code. Shared code also may come from functions with multiple entry points. Two possible representations of functions with multiple entry points are one function with multiple entry points or multiple single-entry-point functions that share code. To our best knowledge, no tool uses the first representation. Under the second representation, a common mistake is to assume that a block of code can only belong to one function. We have observed this code construct in libc, code compiled by the Intel Compilers (ICC) and Fortran functions with programmer specified multiple entry points (use of the "entry" keyword).

Non-contiguous function: the basic blocks of a function are not contiguous in memory. Functions in the source code are always contiguous in source files, but this property may not hold true in binary code for a variety of reasons, including the compiler outlining infrequently executed code to increase cache performance. Therefore, we cannot simply represent the function boundary with an interval from the lowest address to the highest address.

Tail call: a tail call [14] is a compiler optimization that uses a jump instruction at the end of a function to target the entry point of another function. The optimization eliminates a stack frame set-up and a stack frame tear-down by replacing a call instruction with a jump instruction. If a binary analysis tool kit cannot identify tail calls, the control flow edge from a tail call jump instruction to the jump target will be wrongly labeled intraprocedural.

3.4 Relations between Analysis Stages

We make two observations on the relations between the three analysis stages. First, there is interaction between code discovery and CFG construction. On one hand, code discovery is a foundation to build the CFG since the nodes consist of blocks of instructions and control flow edges are specified by the instructions. On the other hand, control flow can be used to address the challenges in code discovery: targets of control flow instructions should always be real code, not data or padding bytes; overlapping instructions can be identified by following their incoming control flow.

Second, CFG partitioning is based on code discovery and CFG construction. An important task of CFG partitioning is to determine function entries and exits. It is significantly more difficult to determine function entries when function symbols are missing, incomplete or inaccurate. For function exits, a binary function usually terminates in a return instruction. However, this is not always the case. As we saw above, a tail call (a jump instruction) and a call instruction to a non-returning function also terminate a function.

While this is an interesting list of problematic code constructs, it is by no means complete. As new code generators are produced, and new optimized libraries are produced, there will be new challenging constructs.

4. CODE DISCOVERY

Non-code bytes intermixed with actual instructions, missing symbols, and overlapping instructions all complicate code discovery. We use real code examples to illustrate why these code constructs are challenging and how we address them.

4.1 Non-code Bytes

Non-code bytes such as jump table data, static read-only data and padding bytes often appear in code sections. A code example from libc 2.12 is shown in Figure 1, where a jump table is in code sections. If the jump table is misinterpreted as code, we can inaccurately identify its contents as valid instructions, as shown in the gray shaded cells.

Some existing tool kits use linear scan to discover code [3, 20, 22]. This approach decodes instructions sequentially starting from a specific point, such as the program entry point or known function entry points. In Figure 1, a tool kit that uses the linear scan approach will continue to decode the non-code bytes in the jump table, after the indirect jump at address 80b15ee is decoded. If these non-code bytes correspond to valid instructions, it is difficult to know to stop the scan.

Dyninst uses control flow (recursive) traversal [46, 49] to address non-code bytes. It starts from known function entry points, follows control flow transfers of the program to discover code and identify more function entry points. In the above example, this approach will not misinterpret the jump table as code since the jump table does not have any incoming control flow and will not be discovered as code during the traversal. For stripped binaries, the coverage of code discovery by using control flow traversal depends on the ability to identify missing function entry points (Section 4.2) and resolve indirect control flow (Section 5.1).

4.2 Missing Symbols

The symbol tables of "stripped" binaries have been removed. Function symbols are a major source of data about function entry points, which are basis for accurate and complete code discovery, and determining function boundaries.

One approach to detect function entry points in stripped binaries is based on an observation that functions often have common operations at the entry, such as setting up a stack frame. These common operations result in common instruction sequences. If we can learn these sequences, we can find function entry points. The above observation leads to a pattern matching based approach that uses a small number of manually designed instruction patterns [21, 22, 25, 35, 47]. However, this approach has been shown to be insufficient because it cannot adapt to variations in compilers and optimization levels [5, 43].

Recent work has used supervised machine learning techniques to learn features for identifying function entry points [5, 43]. Dyninst uses Rosenblum et al.'s method [43] to identify function entry points. Their approach extracts instruction sequences from a training set of binaries and assigns each instruction sequence a weight to represent the probability that an address is a function entry point when the instruction sequence is matched at the address. We applied Rosenblum et al.'s method [43] to train a new model based on the binary code data set published by Bao et al. [5] and got similar entry point identification results to theirs.

4.3 Overlapping Instructions

Overlapping instructions are often seen in malware. Figure 2 shows such an example, where three sequences of blocks overlap and all three sequences will execute at some point in the malware. However, we also observed this code construct in conventional code. As shown in Figure 3, two instructions overlap in this code example from libc-2.12.so. When the program is multi-threaded, the program executes Sequence 1. When the program is single-threaded, the instruction in Sequence 2 is executed; in this case, the `lock` prefix is omitted to avoid the locking overhead.

SecondWrite [47] treats jumping into the middle of an instruction as an invalid case, thus it cannot handle overlapping instructions. Dyninst drops the constraint and follows control flow transfers to report overlapping instructions.

5. CFG CONSTRUCTION

Indirect control flow and non-returning functions complicate construction of the CFG. For indirect control flow, we focus on precisely resolving jump tables. Previous tools used one of the following three approaches to handle jump tables:

Address	Byte	Sequence 1	Sequence 2	Sequence 3
454017	b8			
454018	eb	mov eax, ebb907eb		
454019	07			
45401a	b9			
45401b	eb		jmp 45402c	
45401c	0f			
45401d	90	seto bl		
45401e	eb			jmp 454028
45401f	08	or ch, bh		
454020	fd			

Figure 2: An example of overlapping instructions from a piece of malware. All three sequences of blocks execute.

Address	Byte	Sequence 1	Sequence 2
3fe9e8	74	je 3fe9eb	
3fe9e9	01		
3fe9ea	f0	lock cmpxchg %ecx, 0x35b0(%ebx)	
3fe9eb	0f		cmpxchg %ecx, 0x35b0(%ebx)
...	..		
3fe9f1	00		

Figure 3: An example of overlapping instructions from libc. The instruction starting at address 3fe9ea overlaps with the instruction starting at address 3fe9eb.

(1) deep analysis that can analyze all indirect control flows [4, 6, 28, 29], (2) compiler-specific patterns to identify jump tables [21, 26], or (3) principled jump table analysis based on limited definitions for jump tables [13]. The first approach can handle all types of jump tables, but in many cases will report imprecise control flow targets of jump tables. The second and third approach can precisely resolve some specific types of jump tables, but will fail to resolve new types of jump tables introduced by modern compilers.

5.1 Jump Tables

Our handling of jump tables is based on a new model of jump tables and a dataflow analysis that implements the model. We first present our modeling of jump tables. Our model abstracts jump table calculation as a univariate function that calculates the jump target, which we call *jump table target function*. A jump table target function has several *jump table parameters*, including the contents, location, and size of the table. To statically resolve a jump table, it is essential to analyze the code to determine the form of the jump table target function and to extract the values of the jump table parameters. We present three jump table examples to explain how our model can be implemented. Finally, we briefly discuss our analysis that improves on our ability to populate our model and resolve jump tables.

Jump tables vary mainly in four dimensions: whether the table contents are jump target *addresses* or jump target *offsets* relative to a base address, whether the location of the table is explicitly encoded in an instruction or computed, whether the input to a jump table is bounded by conditional jumps or bounded by computation, and the number of levels of tables involved in the address calculation. Our model for a jump table is split into the following pieces to capture these variations. First, we define the one-level jump table function JT that abstracts reading values from a one-level table. Next, we define the t-level jump table function JT_t that abstracts how multiple one-level jump table functions can be composed to form a t-level table. Last, we define the jump table target function JTT that calculates the control flow target using the values returned by the JT_t.

Instruction	Jump target analysis
`mov %ebp,0xf8(%rsp)`	0 ≤`%rdx==0xf8(%rsp)==%ebp`≤ 5
`cmp $0x5,%ebp` `ja 43a4ab`	0 ≤`%ebp`≤ 5
`lea 0x525e8f(%rip),%rax` `lea -0x302(%rip),%rcx`	`%rax=0x9602a0`, `%rcx=0x43a116` JTT=`0x43a116`+[`0x9602a0`+`%rdx`×8]
`movslq 0xf8(%rsp),%rdx`	`%rdx==0xf8(%rsp)`
`add (%rax,%rdx,8),%rcx`	JTT=`%rcx`+[`%rax`+`%rdx`×8]
`jmpq *%rcx`	JTT=`%rcx`

Figure 4: A one-level jump table from MySQL on Linux. The third column shows how our backward dataflow analysis resolves the jump table. JTT represents the jump table target.

Definition 4 (One-level jump table function) We define $JT_{E,T}(x)$ as the value read from a one-level table when the input is $x \in [l, u]$, where

l and u are the lower and upper bounds of the input to the jump table. We extract the values of l and u so that we can identify all the values in the table. We have identified three scenarios where we can determine the values of l and u. First, when there exists an explicit bounds check, such as a pair of `cmp` and conditional-jump instructions. Second, when the bounds can be inferred from an instruction that operates on the input. For example, the instruction "`and $0xf,%eax`" guarantees that `%eax` is in the range [0,15]. If `%eax` is then used as the input to the jump table, we can infer that $l = 0$ and $u = 15$. Third, in a multi-level table, the values of the earlier tables are used as input to the later tables. Note that in the third case, since these values are statically determined, the compiler does not need to generate instructions to bounds-check the access to the later level tables. Therefore, to determine l and u, We must take into account the contents of the earlier tables.

$E = \{(a_0,v_0),(a_1,v_1),\ldots,(a_{n-1},v_{n-1})\}$ is a set that represents the contents of a one-level jump table, where a_i is the address of the ith table entry and v_i is the value. The table entries are of equal size and laid out contiguously in memory, so the table stride (the distance between adjacent entries), $a_i - a_{i-1}$, are all equal. Specifically $a_i = a_0 + i \times (a_1 - a_0)$, $i \in [0, n-1]$. To extract the value of a_0 and the stride from the code, we identify the instructions that calculate the address of a table entry, convert these instructions to abstract syntax trees (ASTs), and combine these ASTs into a single AST that represents the address calculation of a table entry.

T represents the data type of the values in the table. T specifies the width of the read from the table and whether the read value is signed or unsigned. For example, on a 64-bit Pentium processor, T can be one of several types, including unsigned or signed with sizes ranging from 1 to 8 bytes. Whether the values are unsigned or signed can be determined by checking the opcode of the memory access instruction, and the read width can be determined by the size of the memory operand.

$JT_{E,T}(x) = R_{E,T}(C(x))$ is composed of two functions, where $C(x)$ is a function that calculates which table entry to read, and $R_{E,T}(i) = (T)v_i, i \in [0, n-1]$ represents reading that entry, treating the value as type T. Here we use the C-style type-cast notation to denote that the value of a table entry is converted into data type T. $C(x) = x, x \in [0, n-1]$ is a common form, meaning that the input to the table is directly used to index the table. $C(x)$ can also be in other forms. For example, $C(x) = x >> 2, x \in [0, 4n-1]$ means that the input values are clustered into groups in size four before indexing the table.

Instruction	Jump target analysis
`movzbl (%rdi),%eax`	0 ≤`%eax`≤ 255
`shr $0x4,%al`	`rax`= `rax` >> 4 JTT= [`0x495e30`+(`%rax`>> 4)×8]
`jmpq *0x495e30(,%rax,8)`	JTT= [`0x495e30`+`%rax`×8]

Figure 5: A one-level jump table from Binutils on Linux. The input upper bound to this jump table must be inferred. In addition, the input is right shifted to get the index into the table.

Instruction	Jump target analysis
`cmp $0xa9,%eax` `ja 0x41677e`	0 ≤`%eax`≤`0xa9`
`movzbl 0x416bd4(%eax),` `%ecx`	`%ecx`=[`0x416bd4`+`%eax`] JTT=[`0x416bc0`+[`0x416bd4`+`%eax`]×4]
`jmp *0x416bc0(,%ecx,4)`	JTT=[`0x416bc0`+`%ecx`×4]

Figure 6: A two-level jump table from PSFTP on Windows.

Definition 5 (t-level jump table function) Given t one-level tables, the jump table functions are composed in the expected way: $JT_t(x) = (JT_{E_1,T_1} \circ \cdots \circ JT_{E_t,T_t})(x)$.

Definition 6 (jump table target function) We define $JTT_{t,jb,js}(x) = jb + js \times JT_t(x)$, $x \in [0, n-1]$ as the final jump target, where jb is the jump target base and $js = \pm 1$. When $jb = 0, js = 1$, the t-level jump table contains absolute addresses; when $jb \neq 0$, the t-level jump table contains offsets relative to the base address jb.

To determine t, we need to identify each level of the t-level table and track how the values from an earlier table are used as input to a later table. To determine jb and js, we identify the instructions that use the value read from the t-level table to calculate the jump target and produce an AST that represents the jump target calculation.

We use three examples to explain how to apply our model to real code. The first example is a one-level jump table from MySQL 5.6.3 on x86-64 Linux compiled by ICC 13.0.1, shown in Figure 4. Here, three variables are aliased to the table upper bound u, making it difficult to identify its value. We must make the following three observations to determine that $u = 5$: based on the `add` instruction, `%rdx` represents which table entry to read; based on the `mov` and `movslq` instructions, `%rdx`, `%ebp` and `0xf8(%rsp)` are aliased to each other; and based on the `cmp` and `ja` instructions, `%ebp` is in [0,5]. Note that besides a `cmp` instruction, other instructions that set flags can also specify the value of u, such as `sub`.

The second example is a one-level jump table from Binutils 2.23 on x86-64 Linux compiled by GCC 4.7.2, shown in Figure 5. In this example, we make two observations. First, the upper bound must be inferred as the code does not have an explicit upper bound check. The `movzbl` instruction reads a one-byte value and zero extends it into `%eax`, so `%eax` is in the range [0,255]. Second, the input to the jump table can be grouped before indexing the table. The input value `%eax` is right shifted for four bits and used to index the table. Therefore, $C(x) = x >> 4, x \in [0, 255]$.

Our last example is a two-level jump table from PSFTP 0.58 on x86 Windows compiled by Microsoft Visual Studio 2013, shown in Figure 6. In this example, knowing the contents of the first level table avoids the need for a bound check on the second level table lookup. The `movzbl` instruction

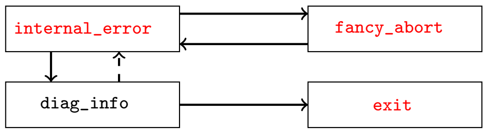

Figure 7: Non-returning functions example from GCC. A node represents a function. A solid edge represents a function call and a dashed edge represents returning to its caller. Non-returning functions are marked in red.

reads from the first level table into `%ecx` and all the contents in the first level table are in the range [0,4]. `%ecx` can then be directly used to index the second level table, without an explicit bound check.

We designed a backward intraprocedural dataflow analysis to derive our jump table model from binary code. Our analysis performs backward slicing on every indirect jump and analyzes the instructions in the slice to populate the model. The analysis first determines jb and js to understand whether the jump table is of absolute addresses or relative offsets, then identifies how many levels of tables are involved and determines the locations and contents of each level table. Finally, the analysis determines the input lower and upper bounds. The slicing can be stopped as soon as either the indirect jump is resolved or we are sure that the jump does not represent a jump table.

5.2 Non-returning Functions

Not being able to identify non-returning functions introduces bogus control flow edges. Previous tools either assume that all functions return to their call sites [29] or use a simple name matching method to identify non-returning functions [5, 21]. This name matching method checks whether the callee of a function call is in a list of well-known non-returning functions, including `exit` and `abort`. If the callee is in the list, the function call will never return. Such a list often includes only the non-returning functions in well-known libraries, such as libc. This name matching method often can be effectively applied to stripped binaries. The key is to determine if a function call targets a known non-returning function. For dynamically linked stripped binaries, the symbols of imported functions are retained to support linking so that the names of the target in calls to dynamically linked libraries are known; for statically linked stripped binaries, library fingerprinting can be used to identify which library function is being called [24].

Bao et al. [5] describe an improvement over this name matching method by noting that if function f always calls function g, and g is identified as a non-returning function, f should also be considered as a non-returning function.

Figure 7 shows an example from GCC 4.9.2 itself compiled by GCC 4.4.7, where Bao et al.'s technique would fail to identify two non-returning functions that are mutually recursive. In this example, `fancy_abort` and `internal_error` are two functions that are mutually recursive. Once the program enters `fancy_abort` or `internal_error`, the program could either reach `exit` or an error would happen due to stack overflow. In either case, `fancy_abort` and `internal_error` will not return to their callers, so they are non-returning functions. When applying Bao et al.'s technique to this example, we first note that neither `internal_error` nor `fancy_abort` is a known non-returning function. We then find that `internal_error`

input : F: a set of functions; and *knownNonRet*: a set of known non-returning functions
output : *nonRet*: a set of identified non-returning functions

1 *nonRet* ← *knownNonRet* $\cap F$;
2 *ret* ← $\emptyset$;
3 *funcList* ← $F-$ *nonRet*;
4 *oldList* ← $\emptyset$;
// Fix point calculation
5 **while** *funcList* $\neq$ *oldList* **do**
6   *oldList* ← *funcList*;
    // Inspect all "unknown" functions
7   **for** $f \in$ *funcList* **do**
8   *blocks* ← ReachableBlocks(f, *funcList*);
        // If f has a return block or tail calls a returning function, it is a returning function
9   **if** ContainRetBlock(*blocks*) **or** TailCall(f, ret) **then**
10   *ret*←*ret*$\cup\{f\}$;
        // If none of the control flow paths returns, f is a non-returning function.
11   **if** NoBlockedCalls(*blocks, funcList*) **and** $f \notin$ *ret* **then**
12   *nonRet* ← *nonRet* $\cup\{f\}$;
    // Determine the functions to be revisited
13   *funcList* ← $F-$*nonRet*$-$*ret*;
// Resolve cyclic dependencies
14 *nonRet* ← $F-$*ret*;

Figure 8: Non-returning function analysis.

always calls `fancy_abort`, but we cannot conclude that `internal_error` is a non-returning function without knowing `fancy_abort` is non-returning. Similarly, we cannot conclude that `fancy_abort` is a non-returning function without knowing `internal_error` is non-returning. Therefore, this technique would fail because the two non-returning functions form cyclic dependencies.

We have designed an interprocedural analysis to determine all the non-returning functions in a program, shown in Figure 8. We use a fix point calculation to detect cyclic dependencies. Note that once the program enters any function in the cyclic dependencies, the program could reach identified non-returning functions, or would stay in the cycle until the stack is overflown. So, all involved functions can be marked non-returning. Our analysis takes as input the set of functions F in the program and a set of known non-returning functions `knownNonRet`; the analysis outputs the set of identified non-returning functions `nonRet`. We calculate a return status for each function. The return status can be "unknown," "might return" or "does not return." The sets `nonRet` and `ret` represent the currently identified "does not return" and "might return" functions, respectively. Initially, all functions have "unknown" return status.

At the beginning of our analysis, all functions in `knownNonRet` are set to "does not return" (line 1). We then perform a fix point calculation to determine the return status of all the other functions in F (lines 5-13). After we reach a fix point, it is possible that there exist cyclic dependencies between the functions whose return status remain "unknown". We set all of them to be "does not return" (line 14).

In each round of iteration, we try to determine the return status of functions in `funcList`, which is a set of functions that currently have "unknown" return status. We define four subroutines: (1) `ReachableBlocks`(f, `funcList`) calculates a set of reachable blocks from the entry node of function f by traversing only known intraprocedural edges. If f calls $g \in$ `funcList`, we are not certain whether the control flow will return from g. Therefore, we do not assume the existence of a call fall-through edge (line 8). (2) `ContainRetBlock(blocks)` returns `true` if one block in `blocks` is a return block. In

```
35110db510 <__write>:
35110db510   cmpl $0x0,0x2b8199(%rip)
35110db517   jne 35110db529
35110db519 <__write_nocancel>:
35110db519   mov $0x1,%eax
...
35110db526   jae 35110db559
35110db528   retq
35110db529   sub $0x8,%rsp
...
35110db556   jae 35110db559
35110db558   retq
35110db559   mov 0x2b2a48(%rip),%rcx
...
35110db56c   jmp 35110db558
```

Figure 9: Functions sharing code and non-contiguous functions example from libc. The code in blue is shared by both functions. `__write_nocancel` is also a non-contiguous function, which is separated by the code from `__write`.

this case, we are sure f is a returning function (lines 9-10). (3) `TailCall(f, ret)` return `true` if f performs a tail call to an identified returning function (Section 6.2). In this case, f is also a returning function (lines 9-10). (4) `NoBlockedCalls(blocks, funcList)` returns `true` if no block in `blocks` calls or tail calls any function in `funcList`. In this case, the function boundary of f and the return status of the tail callees of f have been determined. If we have not marked f as a returning function yet, then f must be a non-returning function (lines 11-12).

6. CFG PARTITIONING

Functions that share code, functions that are laid out non-contiguously in memory, and tail calls make it difficult to partition the CFG into separate functions. The challenge is to produce a partitioning that is consistent with the binary code and that maps reasonably to source code.

6.1 Complex Functions

Function sharing code: code blocks can be shared by multiple functions. Figure 9 shows an example from libc-2.12.so, where two functions sharing code. In this example, `__write` and `__write_nocancel` both are entry points of system call `write`. `__write` supports multithreading, while `__write_nocancel` does not.

Three existing tools allow functions to share code [5, 21, 11]. BYTEWEIGHT represents a function as a set of bytes and allow functions to have common bytes [5]. BAP and Dyninst adopt a definition for functions similar to that of Harris and Miller [21], which allows functions to share code. Specifically, if function f_1 has code blocks V_1 and function f_2 has code blocks V_2, $V_1 \cap V_2$ can be a non-empty set.

Non-contiguous functions: code from other functions can make a function non-contiguous. Figure 9 also serves as an example of non-contiguous functions, where code of `__write_nocancel` is separated by code from `__write`.

As mentioned above, BYTEWEIGHT represents function as a set of bytes. They explicitly point out that the bytes do not have to be contiguous [5]. BAP and Dyninst represent the code of a function as a set of basic blocks and the blocks can be separated by any bytes.

6.2 Tail Calls

A tail call replaces a call instruction with a jump to eliminate a stack frame set-up and tear-down. A simple strategy

```
BZFILE* BZ_API (BZ2_bzdopen) (int fd, char * mode)
 { return bzopen_or_bzdopen(NULL,fd,mode,1);}
BZFILE* BZ_API (BZ2_bzopen) (char *path, char * mode)
 { return bzopen_or_bzdopen(path,-1,mode,0);}
```
```
// entry point of bzopen_or_bzdopen, but no function symbol
351f40baa0   mov %rbx,-0x30(%rsp)
...
351f40bd70 <BZ2_bzdopen>:
351f40bd70   mov %rsi,%rdx          // set mode
351f40bd73   mov $0x1,%ecx          // set open_mode
351f40bd78   mov %edi,%esi          // set fd
351f40bd7a   xor %edi,%edi          // set path
351f40bd7c   jmpq 351f40baa0
...
351f40bd90 <BZ2_bzopen>:
351f40bd90   mov %rsi,%rdx          // set mode
351f40bd93   xor %ecx,%ecx          // set open_mode
351f40bd95   mov $0xffffffff,%esi   // set fd
351f40bd9a   jmpq 351f40baa0
```

Figure 10: A tail call example from bzip2. `BZ2_bzdopen` and `BZ2_bzopen` both perform a tail call to the internal function `bzopen_or_bzdopen`, which does not have a function symbol.

for identifying tail calls is to treat jumps that target function symbols as tail calls. However, this strategy does not work even for non-stripped binaries, when the compiler does not generate the expected symbols. Figure 10 shows an example from libbz2.so.1.0.4 on RedHat 6 Linux, in which `BZ2_bzdopen` and `BZ2_bzopen` perform tail calls to `bzopen_or_bzdopen`. The compiler did not generate a symbol for `bzopen_or_bzdopen`. As a result, the tool kit must rely on heuristics to sensibly parse the code, either labeling `bzopen_or_bzdopen` as code shared by the other two functions or, more preferably, as a tail-called function. In such situation, there could be more than one consistent and reasonable semantic mapping between source code and binary code.

Existing tools often use a two-step approach to identify tail calls [21, 47]. In the first step, if the jump target is a known function entry point, it is a tail call. In the second step, tools may use different heuristics to identify tail calls when the jump target is not a known function entry point. SecondWrite [47] treats a jump instruction as a tail call if there is a known function between the address of the jump instruction and the address of the jump target. Note that SecondWrite's treatment for tail calls implies that they do not allow code for a single function to be separated by code from one or more other functions.

Dyninst's current handling of tail calls uses a variation on the two-step approach. In the first step, we use the function entry points reported in the symbol tables and the ones we identified during control flow traversal to check tail calls. In the second step, we rely on two heuristics to identify tail calls and avoid false positives. First, if we can detect stack frame tear-down before a jump instruction, the jump is a tail call. This heuristic is based on the following observation. If function f tail calls g, then the control flow will not come back to f from g. So, f should clean up its stack frame before performing a tail call to g. Second, if we have strong evidence that the jump instruction and the jump target are in the same function, the jump is not a tail call. For example, branch-not-taken edges and call fall-through edges are always intraprocedural. Suppose we discover a jump instruction in function f. If the jump target can be reached from the entry of f by going only through intraprocedural edges, the jump is not a tail call.

7. EVALUATIONS

The eight challenging code constructs introduced in the previous sections were the basis for evaluations of existing binary analysis tool kits, including BAP 0.9.9 [11], GNU Objdump 2.20 [20], IDA Pro 6.6 [22], Jakstab 0.8.3 [25], OllyDbg 2.0.1 [35], SecondWrite (results from SecondWrite group dated 2014-08-17) [47], and our own Dyninst 9.0 [37]. We started our evaluation by performing an extended version of evaluations used by previous researchers. The goal of these evaluations is to answer two questions: (1) Are the challenging code constructs prevalent in real software? (2) Do these binary analysis tool kits perform well in identifying the challenging code constructs? Previous researchers have used real software to evaluate the effectiveness of their techniques on indirect control flow [6, 13, 16, 26, 40, 47] and coverage of code in code sections [13]. We added to these evaluations additional code constructs we identified to better test the effectiveness of the tools. However, these studies are intrinsically limited because we do not know whether a tool kit misses real code constructs (false negatives) or reports bogus code constructs (false positives).

To complement our evaluations, we constructed a controlled experiment by using small hand-crafted programs, which is also a commonly used evaluation strategy [49, 6] and has the advantage of knowing the ground truth through human inspection and verification. We produced test binaries by patterning them after the challenging code constructs we found in real software and evaluate how each of the above tool kits handled the code constructs. These test cases represent precisely the hardest cases we found when evaluating our tool on real software. These test cases help identify these difficult code constructs in a low noise environment.

7.1 Real Software Experiment

In this experiment, we compiled SPECint 2006 using two compilers (GCC 4.4.7 and ICC 15.0.1) with four optimization levels (from -O0 to -O3) on RedHat Linux 6.6. The test binaries are statically linked to include highly optimized library code. Being able to analyze library code is important because library code may account for a large fraction of code executed. The results are shown in Table 2. We present the results for GCC and ICC together as we do not observe significant differences between the results for the two compilers when comparing their minimal, median, and maximal numbers for each code construct. We also repeated the experiments for SPECfp 2006 using GCC 4.4.7 with -O2. The results are basically the same as for SPECint.

The results show that the challenging code constructs are prevalent in real software. Dyninst reported that all the eight code constructs were found in every test binary. The results of BAP and IDA Pro confirmed the prevalence of four code constructs. For the other code constructs for which either BAP or IDA Pro reported nothing, we confirmed by hand that the instances reported by Dyninst are indeed true. Since we lack ground truth, we cannot directly compare the tool kits' capabilities in handling these code constructs. To attempt to explain why tool kits reported significantly different results, we resorted to manual inspection of the results and inspected about ten to twenty randomly sampled instances of each code construct. First, IDA Pro reported more code in code sections than Dyninst. In many cases, it appears that IDA Pro misinterpreted data as real code. In other cases, IDA Pro speculatively disassembled and reported instructions even though it did not know how these instructions could be reached; Dyninst did not report these instructions. Second, we found that all tools reported about the same number of indirect jumps, though Dyninst could resolve the most of these jumps because of our new jump table model. We inspected some of the remaining unresolved indirect jumps from Dyninst and found that they were all indirect tail calls that did not use jump tables. Third, IDA Pro reported many functions without symbols, but many of the reported functions were marked failed, leaving its results difficult to interpret. Fourth, IDA Pro sometimes wrongly classified a function as non-returning function if the function ends with a jump (a tail call) to another returning function. Fifth, BAP did not report any tail calls, which might explain why BAP reported many more groups of functions sharing code and non-contiguous functions than Dyninst. When BAP fails to identify a tail call and treats the jump instruction as intraprocedural, it wrongly reports that the tail-caller and the tail-callee share code. In addition, if another function was between the tail-caller and the tail-callee in memory layout, BAP would wrongly report the tail-caller as a non-contiguous function. Finally, IDAPro reported more tail calls than Dyninst on average. However, many of the tail calls reported by IDAPro are not real. Note that IDAPro failed to report any non-contiguous functions; a jump from one code block to another far away of the same non-contiguous function is wrongly reported as a tail call.

In summary, this experiment shows that the challenging code constructs are prevalent in real software. However, it is difficult to precisely calibrate how well these tool kits performed in identifying these code constructs due to lack for ground truth for the test binaries.

7.2 Test Suite Experiment

To compare tool kits' capabilities in a low noise environment, we constructed test cases by patterning them after the challenging code constructs found in real software including Binutils, bzip2, GCC, and MySQL.

Code discovery: We have three test cases for the code construct non-code bytes, where static read-only data, jump table data (as shown in Figure 1), and padding bytes are embedded in the code sections. A tool kit passes a test when (1) the non-code bytes are not interpreted as code; (2) the last instruction before the non-code bytes is reported; and (3) the first instruction after the non-code bytes is reported.

We strip our test binaries to create the missing symbols test cases. Before we strip the test binaries, we record all function entry points in the symbol table as ground truth. In this test, we report the number of identified real entry points, the total number of real entry points, and the number of identified bogus entry points. We have one test case for overlapping instructions (Figure 3). A tool kit passes if it reports both instructions.

CFG construction: We have six test cases of the abilities of tool kits to resolve indirect control flow; five of these are jump tables. The first test case is a basic jump table, where the input to the jump table is checked by a `cmp` instruction and a conditional jump, and then directly used to index the table. The second test case avoids the bound check by using an `and` instruction. The third to the fifth cases correspond to the examples in Figures 4–6. The sixth test case does not involve a jump table; it is an indirect jump used to handle parameter passing in a function with a variable number of

Table 2: Reports from existing binary analysis tool kits. Each report item reflects how often a corresponding code constructs appear in binaries. We summarize the results by showing the minimal, median, and maximum numbers.

Report item	BAP			IDA Pro			Dyninst		
	Min	Median	Max	Min	Median	Max	Min	Median	Max
Fraction of code in code sections	0.6933	0.7583	0.9061	0.9311	0.9621	0.9954	0.9298	0.9514	0.9940
Fraction of resolved indirect jumps	0.0000	0.0059	0.1148	0.0556	0.2549	0.6829	0.1637	0.7532	0.9377
# of functions without symbols	5	7	50	65	71	244	6	6	34
# of groups of overlapping instructions	0	0	0	0	0	0	16	16	17
# of non-returning functions	1	2	11	43	54	496	13	18	447
# of groups of functions sharing code	430	485	4113	0	0	0	12	13	31
# of non-contiguous functions	354	407	6573	0	0	0	61	63	69
# of tail calls	0	0	0	710	790	2691	200	251	6745

Table 3: Evaluation results of existing binary analysis tools. For missing symbols, a result entry in the form of "x/y,z" means that the tool correctly recovered x out of y total functions, but also reported z bogus functions. In all other cases, a result entry contains a string composed of `P` (Pass), `p` (Pass, but less preferable), `F` (Fail), `X` (eXit abnormally), `-` (Not applicable); the length of the string represents the total number of test cases; the ith character in the string represents the result of the ith test.

Stage	Code construct	Arch	BAP	Objdump	IDA Pro	Jakstab	OllyDbg	SecondWrite	Dyninst
Code discovery	Non-code bytes	32-bit	PFP	FFF	FPP	FXP	P-P	PPP	PPP
		64-bit	PFP	FFF	FPP	XXX	X-X	XXX	PPP
	Missing symbols	32-bit	68/93,2	0/93,0	35/93,0	X	-	20/93,14	86/93,0
		64-bit	879/1163,49	0/1163,0	608/1163,408	X	-	X	1080/1163,60
	Overlapping instructions	32-bit	P	F	P	X	-	X	P
		64-bit	P	F	P	X	-	X	P
CFG construction	Indirect control flow	32-bit	FFFFFF	FFFFFF	PFFFPF	FXXXFX	P---F-	FXXXXX	PPPPPP
		64-bit	FFFFFF	FFFFFF	PFFFPF	XXXXXX	X---X-	XXXXXX	PPPPPP
	Non-returning functions	32-bit	FF	FF	PF	PP	FF	PF	PP
		64-bit	FF	FF	PF	XX	XX	XX	PP
CFG partitioning	Functions sharing code	32-bit	P	F	F	X	-	X	P
		64-bit	P	F	F	X	-	X	P
	Non-contiguous functions	32-bit	P	F	F	X	-	X	P
		64-bit	P	F	F	X	-	X	P
	Tail calls	32-bit	Fpp	PFF	PPP	PFX	---	PPP	PPP
		64-bit	Fpp	PFF	PpF	XXX	---	XXX	PPp

arguments, such as printf. A tool kit passes a test if it reports exactly the real control flow targets of the indirect jump.

We designed two test cases for non-returning functions: (1) a function calls `exit` at the end, and (2) two non-returning functions are mutually recursive, as shown in Figure 7. A tool kit passes if it reports all the non-returning functions.

CFG partitioning: We have one test for functions sharing code, as shown Figure 9. A tool kit passes if it reports that the shared code is in both functions. The above test is also used for non-contiguous functions. A tool kit passes if it reports all the code of the non-contiguous function.

There are three tail call test cases. The first test is a basic case where a function performs a tail call to another function, and the callee has a defined function symbol. A tool kit passes the test if the tail call is correctly identified. The second test is where two functions do recursive tail calls to each other. Neither function has a corresponding function symbol. The third test is where two functions perform tail calls to a third function, as shown in Figure 10. For the second and third tests, a tool kit passes if it correctly identifies the tail calls or if it reports a consistent CFG partitioning, where two functions share code without reporting the tail-called function. Note that both partitioning results are semantically correct. We denote the former one with `P` and the later one with `p`, representing that the former one is likely more preferable than the latter one.

Evaluation results: The results are presented in Table 3. For JakStab and SecondWrite, the results for the 64-bit tests are all "X" because they do not support 64-bit binaries; some entries for 32-bit tests are "X" due to instruction decoding errors. OllyDbg only supports 32-bit Window binaries, so some of our tests were not applicable to it.

8. CONCLUSION

We have presented challenging code constructs generated by modern compilers that makes binary code analysis more difficult. These challenging code constructs complicate code discovery (finding all instructions in a program), building an accurate (or, at least, plausible) CFG for the program, and CFG partitioning (determining function boundaries). We described Dyninst's new code parsing algorithms to handle these new constructs, including a new model for describing jump tables that improves our ability to precisely determine the control flow targets, a new interprocedural analysis to determine when a function is non-returning. and techniques for handling tail calls, code overlapping between functions, and code overlapping within instructions.

We used real-world code examples to illustrate each code construct and discuss the approach used in Dyninst to handle each construct. Our evaluation then compared Dyninst to other available binary tool kits to show their effectiveness in correctly interpreting these code constructs. In all cases, Dyninst was able to accurately parse these examples, while the other tool kits all had significant limitations.

9. ACKNOWLEDGEMENTS

We appreciate the coding help and feedback on the prose from Emily Gember-Jacobson and Bill Williams. This work is supported in part by DOE grant DE-SC0010474, NSF Cyber Infrastructure grants OCI-1032341, OCI-1127210, OCI-1234408, and the DHS under AFRL Contract FA8750-12-2-0289.

10. REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, Apr. 2010.

[2] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, Long Beach, California, USA, March 2007.

[3] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86: A Platform for Analyzing x86 Executables. In *14th International Conference on Compiler Construction (CC)*, pages 250–254, Edinburgh, UK, 2005.

[4] G. Balakrishnan and T. Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Transactions on Programming Languages and Systems*, 32(6):23:1–23:84, Aug. 2010.

[5] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *23rd USENIX Conference on Security Symposium (SEC)*, pages 845–860, San Diego, CA, Aug. 2014.

[6] S. Bardin, P. Herrmann, and F. Védrine. Refinement-based cfg reconstruction from unstructured programs. In *12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 54–69, Austin, TX, USA, Jan. 2011.

[7] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE)*, pages 9–16, Szeged, Hungary, Sep. 2011.

[8] A. R. Bernat and B. P. Miller. Structured Binary Editing with a CFG Transformation Algebra. In *2012 19th Working Conference on Reverse Engineering (WCRE)*, Kingston, Ontario, Canada, October 2012.

[9] A. R. Bernat, K. Roundy, and B. P. Miller. Efficient, sensitivity resistant binary instrumentation. In *the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, pages 89–99, Toronto, Ontario, Canada, July 2011.

[10] M. Bourquin, A. King, and E. Robbins. Binslayer: Accurate comparison of binary executables. In *2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, Rome, Italy, Jan. 2013.

[11] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *23rd International Conference on Computer Aided Verification (CAV)*, pages 463–469, Cliff Lodge, Snowbird, Utah, July 2011.

[12] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *17th Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, Feb. 2010.

[13] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *7th International Workshop on Program Comprehension (IWPC)*, Pittsburgh, PA, USA, May 1999. IEEE Computer Society.

[14] W. D. Clinger. Proper tail recursion and space efficiency. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–185, Montreal, Canada, June 1998. ACM Press.

[15] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In *2002 ACM/IEEE Conference on Supercomputing (SC)*, pages 1–11, Baltimore, Maryland, 2002.

[16] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, Nevada, USA, Jun. 2000.

[17] F. C. Eigler and Red Hat, Inc. Problem solving with SystemTap. In *Proc. of the Ottawa Linux Symposium*, Ottawa, Ontario, July 2006. Citeseer.

[18] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 51–60, Seattle, Washington, USA, 2013. ACM.

[19] W. Fang, B. P. Miller, and J. A. Kupsch. Automated tracing and visualization of software security structure and properties. In *Ninth International Symposium on Visualization for Cyber Security (VizSec)*, pages 9–16, Seattle, Washington, 2012. ACM.

[20] GNU Project. GNU Binutils, http://www.gnu.org/software/binutils.

[21] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, Dec. 2005.

[22] Hex-Rays. IDA, https://www.hex-rays.com/products/ida/.

[23] E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller. Detecting code reuse attacks with a model of conformant program execution. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, Munich, Germany, Feb. 2014.

[24] E. R. Jacobson, N. Rosenblum, and B. P. Miller. Labeling library functions in stripped binaries. In *10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE)*, Szeged, Hungary, Sep. 2011.

[25] Jakstab. http://www.jakstab.org/home.

[26] D. Kästner and S. Wilhelm. Generic control flow reconstruction from assembly code. In *Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES/SCOPES)*, pages 46–55, Berlin, Germany, 2002.

[27] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *10th Working Conference on Mining Software Repositories (MSR)*, San Francisco, CA, USA, May 2013.

[28] J. Kinder and D. Kravchenko. Alternating control flow reconstruction. In *13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Philadelphia, PA, Jan. 2012.

[29] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *20th International Conference on Computer Aided Verification (CAV)*, pages 423–427, Princeton, NJ, USA, July 2008.

[30] G. Llort and H. Servat. *Extrae*. Barcelona Supercomputer Center, 2015.

[31] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, United Kingdom, June 2014.

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

[33] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, Nov. 1995.

[34] J. Mußler, D. Lorenz, and F. Wolf. Reducing the overhead of direct application instrumentation using prior static analysis. In *Proceedings of the 17th international conference on Parallel processing-Volume Part I (Euro-Par 2011)*, Bordeaux, France, sep 2011. Springer-Verlag.

[35] OllyDbg. http://www.ollydbg.de.

[36] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting Security in COTS Software with Binary Rewriting. In *26th IFIP TC-11 International Information Security Conference (IFIP SEC)*, pages 154–172, Hamburg, Germany, June 2011.

[37] Paradyn Project. Dyninst: Putting the Performance in High Performance Computing, http://www.dyninst.org.

[38] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: Force-executing binary programs for security applications. In *23rd USENIX Conference on Security Symposium (SEC)*, San Diego, CA, Aug. 2014.

[39] C. Reffett and D. Fleck. Securing applications with dyninst. In *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–6, Waltham, MA, USA, April 2015.

[40] T. Reinbacher and J. Brauer. Precise control flow reconstruction using boolean logic. In *Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 117–126, Taipei, Taiwan, Oct. 2011.

[41] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *2011 International Symposium on Software Testing and Analysis (ISSTA)*, pages 100–110, Toronto, Ontario, Canada, July 2011.

[42] N. Rosenblum, X. Zhu, and B. P. Miller. Who wrote this code? identifying the authors of program binaries. In *16th European Conference on Research in Computer Security (ESORICS)*, Leuven, Belgium, Sep. 2011.

[43] N. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. Learning to analyze binary computer code. In *23rd National Conference on Artificial Intelligence (AAAI)*, pages 798–804, Chicago, Illinois, July 2008. AAAI Press.

[44] K. A. Roundy. Hybrid analysis and control of malicious code, doctoral disstertation, University of Wisconsin-Madison, 2012.

[45] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Open|SpeedShop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2-3):105–121, 2008.

[46] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering (WCRE)*, Richmond, VA, USA, Oct 2002.

[47] M. Smithson, K. Elwazeer, K. Anand, A. Kotha, and R. Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *20th Working Conference on Reverse Engineering WCRE*, pages 52–61, Koblenz, Germany, October 2013.

[48] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1013–1019, Las Vegas, Nevada, USA, June 2000.

[49] H. Theiling. Extracting safe and precise control flow from binaries. In *the Seventh International Conference on Real-Time Systems and Applications (RTCSA)*, pages 23–30, Cheju Island, South Korea, Dec. 2000.

[50] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Denver, Colorado, USA, Oct. 2015.

[51] L. Xu, F. Sun, and Z. Su. Constructing precise control flow graphs from binaries. Technical report, Technical Report CSE-2009-27, Department of Computer Science, UC Davis., 2009.

[52] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *22nd USENIX Conference on Security (USENIX)*, pages 337–352, Washington, D.C., Aug. 2013.

[53] J. Zhou and G. Vigna. Detecting attacks that exploit application-logic errors through application-level auditing. In *20th Annual Computer Security Applications Conference (ACSAC)*, pages 168–178, Tucson, AZ, USA, Dec. 2004.