Effective Function Recovery for COTS Binaries using Interface Verification

Rui Qiao and R. Sekar
Stony Brook University
Stony Brook, NY, USA

ABSTRACT

Function recovery is a critical step in many binary analysis and instrumentation tasks. Existing approaches rely on commonly used function prologue patterns to recognize function starts, and possibly epilogues for the ends. However, this approach is not robust when dealing with different compilers, compiler versions, and compilation switches. Although machine learning techniques have been proposed, the possibility of errors still limits their adoption.

In this work, we present a novel function recovery technique that is based on static analysis. Evaluations have shown that we can produce very accurate results that are applicable to a wider set of applications.

1. Introduction

Functions are among the most common constructs in programming languages. While their definitions and declarations are explicit in source code, at the binary level, much information has been lost during the compilation process. Nevertheless, numerous binary analysis and hardening techniques require function information. For reverse engineering tasks such as decompiling [16, 14, 25], function boundary extraction provides the basis for recovering other high level constructs such as function parameters or local variables. In addition, many binary analysis and instrumentation tools are designed to operate on functions. These include binary differencing [15, 11], security policy enforcement [9, 24, 8, 31, 32], type inference [20], in-depth binary analysis such as vulnerability detection [30], and more. Given this common requirement, when function boundaries are not available, the first task of many tools is to identify them [7, 17, 28, 9].

Identifying function boundaries is challenging for stripped COTS binaries since they lack debug, relocation, or symbol table information. In order to identify function boundaries, previous approaches employed a strategy that combines call graph traversal and function prologue pattern matching [1, 7, 17]. Specifically, a binary is first recursively traversed from its entry points (available from binary metadata sections), direct control flow transfers are followed, and any targets of direct call instructions are marked function starts. This procedure ends when no new code is reachable. However, since *indirect* control flow transfer targets cannot be statically resolved, indirectly reachable functions are not identified. To deal with this problem, function prologue pattern matching is typically used for the "gap" area: if commonly used prologue patterns like `push %ebp`; `mov %esp, %ebp` is matched, a function start is identified. The problem with this pattern matching based approach is that it is not robust. For example, different compilers can use different prologue patterns. Moreover, compiler optimizations can move code around therefore predefined patterns may not be matched. The situation is exacerbated when supporting binaries compiled with different compilers, using different compiler versions, or different optimization levels/switches. As a matter of fact, a study has shown that existing tools have an unsatisfactory performance in correctly identifying functions [6].

To overcome the limitation of pattern-matching, machine-learning based approaches have been proposed for function extraction [23, 6, 27]. The idea is to use a set of binaries to train a model for function prologues (and optionally, epilogues). This approach is beneficial because it builds a more complete model, reduces manual efforts, and is able to improve the accuracy to above 95%. However, like other machine learning based approaches, their results are dependent on a good training set, and may require some parameter tuning. More importantly, the recovered boundaries are not accurate enough for many downstream applications. One prominent class is binary instrumentation, which has stringent requirement for the quality of recoverd functions: take the extensively studied control-flow integrity (CFI) [2, 35, 34, 31, 32] as an example, the results produced by above systems cannot be easily used for instrumentation and enforcement because even a single unidentified function start may break the instrumented binary.

In this work, we develop a static analysis based approach for function boundary extraction. Unlike previous approaches, all of which operate on a "best effort" basis, our approach is designed to provide the soundness properties required for several classes on instrumentation applications. Specifically, we make the following contributions:

- *Function identification by checking function interface prop-*

*This work has been submitted to CCS 2016. The title and abstract have been changed in this technical report, but the rest is identical to what was submitted to CCS 2016.

†This work was supported in part by grants from NSF (CNS-1319137) and ONR (N00014-15-1-2378).

DOI: 10.1145/1235

erties. We show that function *interface* properties, as compared to function prologue patterns, can provide valuable evidence for function identification.

- *Systematic approach*. We show that functions can be systematically recovered with a technique that combines function start address enumeration and function interface verification.
- *In-depth evaluation*. We perform comprehensive evaluations with more than 1000 binaries compiled from three different languages, three compilers, and four optimization levels, and have achieved better results than state-of-the-art machine learning based systems with a common dataset.
- *Soundness*. A salient feature of our approach is that the recoverd functions are not only of high quality, but are also sound for many analysis and instrumentation applications.

2. Problem

2.1 Definitions

Program binaries are organized into *sections*. Each section may contain code, data, metadata, or other auxiliary information. A code section consists of a sequence of bytes which is interpreted by the CPU as instructions and gets executed at runtime. There may be metadata about the code sections (and data sections), most notably the *symbol table*, which denotes the symbol type (e.g., function), start offset, and size of each symbol. However, symbol tables are usually stripped off before binaries are distributed.

A function is a sequence of bytes in a code section. These bytes may not be physically contiguous, e.g., there could be *embedded data* in the midst of a function. Functions can be next to each other, or there can be extra *padding bytes* between them.

Our task is to recover bytes belonging to each function. Similar to prior work [6, 27], correctness is determined by matching the start and end address for each function with symbol table information. Note that start and end address are determined by the smallest and largest address of all *bytes* of the function, respectively.

2.2 Function Identification Challenges

Functions may not be directly reachable. Although a fraction of functions in a binary are reachable from direct calls, a significant number of functions are only reachable *indirectly*. Identifying the exact set of function pointer values is an undecidable problem in general, so call graph traversal cannot be used to uncover all of the functions.

Unreachable functions. Note that some functions may simply be unreachable. Such *dead code* may exist for several reasons, e.g., when a compiler inlines a function at every call site. While the recovery of reachable functions is critical for all binary analysis and instrumentation applications, some applications such as binary comparison and forensics require the recovery of unreachable functions as well.

Compiler optimizations. To squeeze performance gains, compilers may generate code in unusual ways. For instance, contrary to the high level abstraction that a function has a single entry point, a function in a binary may have multiple entries. Moreover, instead of being entered via a `call` instruction, tail call optimizations results in the use of `jump`s to enter a function. Because of these, it is challenging to differentiate between intra- and inter-procedural control flow transfers to determine whether the target is a function start.

2.3 Metrics

We use the same metrics, *precision*, *recall*, and *F1* as in previous work [6, 27]. Their definitions are as follows. In these equations, TP denotes the number of true positives for identified functions, FP denotes false positive, while FN denotes false negatives.

$$Recall = \frac{TP}{TP + FN} \tag{1}$$

Note that recall captures the fraction of functions in the binary that are correctly identified by an approach.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

Note that precision represents the conditional probability that a true function has been identified whenever our approach reports a function.

Typically, these two metrics are combined using a harmonic mean into a quantity called F1-score.

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{3}$$

3. Background and Approach Overview

3.1 Disassembly

Disassembly is usually the first step for any binary analysis. There are two major techniques for disassembly: linear sweep and recursive traversal [26]. Each of these techniques has some limitations: linear sweep may erroneously treat embedded data as code, while recursive traversal suffers from completeness problems due to difficulties in statically determining indirect control flow targets.

Recent advances have shown that robust disassembly can be achieved with linear disassembly and error correction mechanisms [35]. More specifically, the disassembly algorithm works by first linearly disassembling the binary, and then checking for errors such as (1) invalid opcode; and (2) direct control transfer outside the current module or to the middle of an instruction. These errors arise due to embedded data and are thus corrected by identifying data start and end locations so that disassembling can skip over them. More details can be found in Reference [35].

Since the disassembly technique has been shown to be correct for a wide range of complicated and low-level binaries [35], in this work, we utilize the same technique for disassembly. One benefit of having correct disassembly is that only instruction beginnings are considered candidates for function starts, rather than every byte in the program text. Moreover, complete disassembly is the basis for in-depth analysis, which provides further information for determining function boundaries.

3.2 Overview of Approach

The key idea of our approach is that of enumerating possible function starts, and then using a static analysis to con-

firm them. From a high level, our approach works by iteratively uncovering definite or possible functions and verifying them as desired. Directly reachable functions are identified first, and then possible indirectly reachable functions are enumerated and checked. Finally, unreachable functions are handled.

Possible function start addresses are enumerated in two ways. Directly reachable functions can be readily enumerated. Moreover, no confirmation is needed for such functions. For indirectly reachable functions, code addresses buried in all binary sections serve as proper function start candidates, while for unreachable functions, the first instructions of unclaimed code regions are considered.

Since functions interact with each other through *interfaces*, we can spot *spurious* functions by checking whether they satisfy properties associated with function interfaces. This includes how control flow can be directed to function starts, and how arguments can be passed.

In the following sections, we elaborate on approaches for enumerating possible function starts, determining function boundaries, and verifying their interfaces.

4. Function Starts

4.1 Definite Start Identification

Functions, as seen from the assembly level, are code snippets that are *called*. Therefore, with the disassembly obtained using the approach described earlier, the targets for direct call instructions are *definite* function starts. They are first collected.

4.2 Possible Start Enumeration

As compared to directly called functions, some functions are only reachable *indirectly*. As a register or memory location is used as the operand, the target is not explicit. Moreover, static analysis cannot resolve the exact set of targets, because it is an undecidable problem.

One property of indirectly reachable functions is that their addresses are taken and stored somewhere in the binary. If we were able to locate these addresses, the start points of these functions can be identified. However, due to lack of debug or relocation information, it is difficult to parse the binary and tell which bytes are function addresses. To address this problem, we consider all *possible* indirectly reachable function start addresses, and then *check* them based on further mechanisms.

The naive approach for function start enumeration is to regard each instruction as a candidate. However, as most instructions in a binary are not function starts, this would give us an unnecessarily large set for further (more expensive) analysis and lead to unduly long analysis time.

Therefore, we need a much smaller yet safe superset. To that end, we choose an approach that is similar to the static analysis in BinCFI [35]. Specifically, we scan through the code and data sections of the binary using an n-byte window to extract constants, where n is the number of bytes for a word in the architecture. For each such constant, we consider it as a candidate for a function start if it satisfies the following properties:

- The constant address falls into the code segments;
- The constant address conforms with instruction boundaries.

The reason why the starts of indirectly reachable functions are included in this set is that their addresses are taken and stored in the binary, and a brute force scan would uncover all of them. Note that this holds for both indirectly called functions and those reached by indirect jumps (i.e., indirect tail calls). Furthermore, as discussed and evaluated in BinCFI, function pointers are typically not involved in pointer arithmetic, therefore the identified constants are a safe superset. On the other hand, as will be shown in Section 7.4.3, this is a much smaller set than that which includes the address of each instruction.

Despite smaller, this candidate set cannot be considered as the exact set of indirectly reachable function starts, as there are many spurious items. We point out these identified constants can at least belong to the following categories:

1. True function pointers;
2. Other code pointers;
3. A byte sequence misinterpreted as function pointer candidate by coincidence.

Therefore, we would need further mechanisms to remove the candidates belonging to the second and third category, which are to be explained in Section 6.

4.2.1 *Unreachable functions*

Other than directly and indirectly reachable functions, there are functions that are not reachable at all. For reasons discussed in Section 2.2, we also try to identify unreachable functions. The basic idea is to analyze the "gap" area, i.e., code regions that are not covered by already identified functions. This procedure is performed *after* the determination of directly reachable functions and verification of indirectly reachable functions.

Because functions may have padding bytes after its end, we consider the first non-NOP instruction in each gap as a potential function start. The CFG is traversed and the potential function end can be identified. If this potential function does not take all the space of the current gap, the remaining region is considered as a new gap, and the process continues until all gaps have been analyzed.

Although our gap exploration seems similar to prior work [1, 17, 7], there are several prominent differences. First, we examine gap areas *after* indirectly reachable functions are determined, therefore the regions left are much smaller. Second, by skipping NOP instructions and embedded data identified by our disassembly algorithm, we increase the likelihood that the starting byte of a gap is a function start. Third, and most important, the identified functions are checked to see if they conform to a function interface.

5. Function Boundaries

Other than function starts, we also want to identify their corresponding ends: obviously, function *boundaries* are the actual desired output. To identify function ends, control-flow graph (CFG) traversal is used. Specifically, from a function entry, all possible paths are followed until control flow exits the function. Note that for conditional jumps, both branches are taken.

Note that although most functions exit with return instructions, there are cases where other control flow types are used. For example, a function may call the exit function

```
08054070 <bfd_fopen>:
8054070:     8b 44 24 04        mov 0x4(%esp),%eax
8054074:     8b 54 24 08        mov 0x8(%esp),%edx
8054078:     8b 4c 24 0c        mov 0xc(%esp),%ecx

0805407c <bfd_fopen.>:
805407c:     56                 push %esi
805407d:     57                 push %edi
805407e:     53                 push %ebx
805407f:     55                 push %ebp
8054080:     83 ec 14           sub $0x14,%esp
8054083:     8b f9              mov %ecx,%edi
8054085:     8b f2              mov %edx,%esi
8054087:     6a 00              push $0x0
8054089:     68 b4 00 00 00     push $0xb4
805408e:     8b e8              mov %eax,%ebp
......
```

Figure 1: An example function with multiple entries

to terminate the program. Mechanisms are required to deal with such exceptional cases.

5.1 Non-return and Tail-Called Functions

To determine non-returning functions, a simple analysis is developed. First, we collect a list of library functions that are documented to never return. We then analyze each directly reachable function of the binary. If it calls known non-return function on each of its control flow path, it is also recognized as a non-returning function and added to the list, and so on.

Identification of tail calls serves two purposes: determining the end of the old function, and detecting start of the new one.

The detection of tail call works by checking each direct jump instruction in the analyzed binary:

1. If the target is a procedure linkage table (PLT) entry or a known function start, it is recognized as a tail call.
2. If the target address is larger than next definite function start, or smaller than the current function start, it is recognized as a tail call.

While the first case is straight-forward, the second case works because jumping beyond the next function start indicates inter-procedural control flow transfer, hence a tail call is identified. In addition, according to our definition in Section 2.1, a function start (and entry) is the smallest address of all its bytes. Hence jumping to an address smaller than current function start also indicates a tail call.

Note that our second case would work correctly with multi-entry functions. Multi-entries are mostly generated by compiler to allow different interfaces into the function. Therefore, the entry snippets are usually different straight-line code prologues which lead to the shared function body, and there is no control flow from an entry backwards to another entry of smaller address. A typical multi-entry function is shown in Figure 1. In this example, `bfd_fopen` and `bfd_fopen.` are two entries of the same function, and the first entry takes arguments from stack while the second one takes arguments from registers.

6. Interface Property Checking

As discussed in Section 4, non-function-starts could be included in our list of possible function start address. Therefore, their derived functions are *spurious* hence need to be removed. We develop function interface verification mechanisms for this purpose.

6.1 Control Flow Properties

Our control flow verification is based on a simple strategy: targets reached by intra-procedural control flow transfers are *not* function starts. Two cases are included: conditional jumps and table jumps.

Conditional jumps are used for implementing two-way intra-procedural branches and therefore not targeting functions. Similarly, table jumps that are typically compiled from switch-case statements are for multi-way branches. Different from these control flows, functions are *called*.

Although conditional jumps and their targets are explicit, situations are different for table jumps. A table jump is basically an indirect jump whose target value originates from a *jump table*, the association with which is not explicit. We therefore developed a static analysis for this purpose. Our approach is similar to previous work [10, 22]: we perform backwards program slicing from each indirect jump instruction, and then compute an expression for the jump target. If the expression matches commonly used table jump patterns, the indirect jump is recognized as a table jump. We then extract the address of jump table and its bound, and collect the target addresses inside that table. Finally, these addresses are removed from the set for enumerated function starts.

Note that we take a conservative strategy for our table jump identification: whenever an indirect jump might not be a table jump, we discontinue its further processing. Moreover, once the jump table bound cannot be precisely determined, we use the smallest value that is safe. This way we will not incorrectly remove true function starts but only keep more spurious ones, which can be checked with further mechanisms.

6.2 Data Flow Properties

As discussed, control flow property verification may not be sufficient for identifying (and discarding) all non-function code pointers. Therefore, we use another interface verification scheme that is based on data flow properties.

6.2.1 *The stack interface*

Our first data flow verification mechanism is concerned with the stack interface: the proper use of return address and arguments is checked. This is because, stack is not only used by functions for local storage, but also for information passing between callers and callees.

Since stack is operated by almost every function, one might consider other stack related properties for function verification. One possible alternative is stack pointer preservation. The observation is that a function usually allocates stack space at its prologue, and deallocates the same size at the epilogue, therefore stack pointer is preserved for the function invocation. On the other hand, if an *internal* instruction is misinterpreted as function start, then when this "function" returns, the original epilogue would likely deallocate stack space that was not allocated. In other words, the stack pointer is not preserved and its value becomes larger

```
0805ce70 <get_date>:  // real function start
805ce70:  55 push %ebp
805ce71:  57 push %edi
805ce72:  56 push %esi
805ce73:  53 push %ebx
......
805d900:  5b pop %ebx // spurious function start
805d901:  5e pop %esi
805d902:  5f pop %edi
805d903:  5d pop %ebp
805d904:  c3 ret
```

Figure 2: Stack pointer is not preserved for a spurious function

(assuming stack grows downwards), hence a spurious function is detected.

One such example is shown in Figure 2. In this code snippet, address `0x805d900` has been identified as a function start candidate. However, its "function" body from `0x805d900` to `0x805d904` indicates that the stack pointer is not preserved (increases by 16).

Although this criteria looks promising, the complication is that stack pointer preservation does not always hold for all functions. One such case is for functions that return a struct: the caller allocates space for the struct (usually on the stack), and pushes its pointer as the implicit first argument and then makes the call; the callee does not explicitly return the struct, but just fills the content using the passed pointer. When the callee returns, the implicit struct pointer is popped by the callee. Therefore, this action would increase stack pointer by 4 (for 32 bit architecture). Another situation is closely related to calling conventions. Although for `cdecl` calling convention (Figure 3) which is common for the UNIX environment on x86-32 architecture, it is the caller that cleans up the stack arguments and callee preserves stack pointer, for other calling conventions such as `stdcall` and `fastcall`, it is the callee that cleans up the stack arguments, therefore leaving the stack pointer to a higher location.

Due to this problem, the simple stack pointer preservation verification is not used. Instead, a more general and calling convention agnostic mechanism is adopted. Specifically, the following properties are verified:

- The return address should only be used by return instructions of the callee.
- The stack arguments and callee-save registers[1] should be properly used. More precisely, before function returns, callee-save registers should be restored to their original values, and not be assigned with stack arguments[2].

Take the spurious function in Figure 2 as an example, the "return address" is popped by instruction `0x805d900` to `ebx`, and the "stack arguments" are saved to callee-save registers `esi`, `edi` and `ebx` before the "function" returns. Hence it violates both rules and fails the verification. Note that these properties are commonly violated by spurious functions, as

[1] `ebx`, `esi`, `edi`, `ebp` are callee-save registers for all common calling conventions.

[2] Scratch registers such as `eax`, `edx` and `ecx` have no such restrictions and are not checked.

Calling convention	Stack cleanup by	Argument passing
cdecl	caller	stack
stdcall	callee	stack
fastcall	callee	ECX, EDX then stack

Figure 3: Arguments passing for different calling conventions

stack allocation and register preservation instructions which "compensate" stack deallocation and register restores appear at early parts of real functions and are typically not included by spurious ones.

6.2.2 *Register arguments and eflags*

Data flows into a function may not be only through stack, but also via registers. However, the `eflags` register is not used as a means for information passing between functions. For further data flow verification, our second mechanism checks if only allowed registers are used for information passing.

Note that the rules for function argument passing are dictated by calling conventions: i.e., whether arguments are passed through stack or registers or both, and in what order they are passed. Figure 3 shows the commonly used calling conventions for UNIX environment. For each calling convention, the first several arguments are passed using the registers listed (if there are any), with the specified order. The remaining ones are passed through stack. The allowed register arguments are derived from the union of all common calling conventions, combined with other factors such as `regparm` attribute of functions[3].

On the other hand, the actual argument passing behavior for a function can be inferred by analyzing its code. A function's register arguments are determined using liveness analysis: if a register is live at "function" entry, we consider it as an argument. This is because, the live register indicates its `use` is before `define` in the function body, therefore it must have been defined before the call and information is passed through it. However, there is an exception: callee-save instructions at function beginning "use" callee-save registers with the purpose of preserving them to stack. Since this does not represent information passing, they should not be considered as *real* uses. Our analysis currently adopts a simple strategy by not considering commonly used callee-save instructions (e.g., `push`%`ebx`) as register uses. With the analysis results, a function is recognized as spurious if a "non-argument" register is detected live at function entry.

Similarly, the information passing behavior through `eflags` could also be analyzed through liveness analysis. Any live flag would indicate intra-procedural data flow between the interface and therefore fail the verification.

7. Evaluation

7.1 Dataset

We used two datasets for evaluation. Our first dataset is the same as that used by most recent works in this area, namely, ByteWeight [6] and the work of Shin et al [27]. Although our approach itself is platform-neutral, our current implementation is limited to x86-32/Linux platform.

[3] The attribute allows the annotated function to pass 1-3 arguments through registers, following the order: EAX, EDX, ECX.

Hence our comparison focuses on the subset of the results for this platform. This dataset consists of 1032 binaries from `binutils`, `coreutils` and `findutils`. They are compiled with GNU (`gcc`) or Intel (`icc`) compilers, from no optimization to the highest optimization. These binaries include 303,238 functions and totaling 138,547,936 bytes, which gives a rough estimate of average function length of 449 bytes.

Our second dataset is the set of SPEC 2006 programs. As compared to the first dataset, which are mostly operating system utilities written in C, SPEC programs are more diverse in terms of their applications, as well as the programming languages used (C, C++, Fortran). To compile these programs, we used the GCC compiler family (`gcc`, `g++`, and `gfortran`) and LLVM (`clang`, `clang++`), with the most commonly used optimization level: `-O2`.

7.2 Implementation

Our main analysis framework is implemented in Python, and consists of about 2900 lines of code. For the disassmbler, we used `objdump` and reimplemented the error correction algorithm from BinCFI [35]. The main framework also includes all major components described, including function start identification, CFG traversal, and control flow analysis, but not data flow interface verification.

The current implementation of the data flow interface verification is based on Jakstab [18] and therefore limited to the x86-32 platform. Jakstab is an analysis platform that performs abstract interpretation on binaries and identifies an over-approaximation of the possible indirect targets, therefore it is able to resolve certain indirect control transfers. However, this technique is limited due to precision loss in the static analysis.

We used Jabstab mainly because it has an extensible data-flow analysis framework. We first modified the interface of Jakstab so that the analysis can work on a specified address range, instead of the whole binary. For stack analysis, we also defined our own abstract domain, which is similar to the one described in Reference [24]. In short, each domain element is a sum of a symbolic base value which denotes the original register value upon function entry, and a constant. The analysis produces at the function end the abstract value of each register and memory location, and how it has changed against the initial value.

Figure 4 shows the initial and end states from our analysis of the example described in Figure 2. In this figure, the capitalized `REG` is the symbolic value denoting the initial value of `reg` upon function entry. The right two columns show the end states for "function" [0x805d900, 0x805d904] and [0x805ce70, 0x805d904], respectively. For "function" [0x805d900, 0x805d904], since callee-saved registers (`ebx`, `esi`, `edi`, `ebp`) derive their value from "stack arguments," it is recognized as spurious. On the other hand, "function" [0x805ce70, 0x805d904] passes stack usage verification. Note that due to the approximations used in the analysis, some register values could be set to `TOP` (i.e., unknown) at the end of a function, but the interface properties can typically be verified despite this.

The same analysis also keeps track of stack locations that have been accessed. The ones above return address are considered as stack arguments and therefore their number can be determined. For liveness analysis, we use the standard backwards data-flow analysis algorithm by computing `gen` and `kill` sets, for registers and `eflags`.

Initial state	End state (for "function" [0x805d900, 0x805d904]	End state (for "function" [0x805ce70, 0x805d904]
ebx = EBX esi = ESI edi = EDI ebp = EBP ...	ebx = *(ESP) esi = *(ESP + 4) edi = *(ESP + 8) ebp = *(ESP + 12) ret_addr = *(ESP + 16) ...	ebx = EBX esi = ESI edi = EDI ebp = EBP ret_addr = *(ESP) ...

Figure 4: The analysis states of example code

Tool	Func start			Func boundary		
	Precision	Recall	F1	Precision	Recall	F1
ByteWeight	0.9841	0.9794	0.9817	0.9278	0.9229	0.9253
Neural	0.9956	0.9906	0.9931	0.9775	0.9534	0.9653
Ours	0.9895	0.9962	0.9928	0.9818	0.9831	0.9824

Figure 5: Function boundary identification results from different tools

7.3 Summary of Results

Figure 5 summarizes function start and boundary identification results for the first dataset. Since the two most recent work [6, 27] achieved good results and outperformed previous tools (such as IDA), we only compare our results with them. Because we tested with the same dataset, we directly use the numbers reported by them.

As shown in the figure, our system produced better function boundary results than previous works. For function start, our results are better than ByteWeight, and comparable to the neural network system: while they achieved better precision, we had higher recall, and the F1-scores are close.

7.3.1 *Segmented Results*

In this section, we present segmented results for our first dataset, based on different compilers and optimization levels. As shown in Figure 6, precision and recall are relatively stable for all optimization levels, but have slight decrease for higher levels. Moreover, our system performs better on `gcc` than `icc` binaries, probably due to the differences in optimization techniques used.

7.4 Detailed Evaluation

7.4.1 *SPEC 2006 Results*

Figure 7 shows the function boundary identification results for our second dataset: SPEC 2006 programs. We can see that our system performance is quite stable across a wide range of applications written in three different languages (C, C++ and Fortran) and compiled with two compilers (GCC and LLVM)[4]. Since prior work [6, 27] did not

[4]LLVM does not have an official frontend for Fortran, therefore the corresponding experiments were omitted, and are denoted as "-" in the figure.

	gcc			icc		
	Precision	Recall	F1	Precision	Recall	F1
O0	0.9848	0.9932	0.9890	0.9841	0.9926	0.9883
O1	0.9986	0.9999	0.9992	0.9828	0.9926	0.9927
O2	0.9964	0.9975	0.9970	0.9512	0.9462	0.9487
O3	0.9954	0.9963	0.9958	0.9511	0.9461	0.9486

Figure 6: Segmented results

Program	Language	Suite	GCC				LLVM			
			Reachable func start		All func boundary		Reachable func start		All func boundary	
			Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
400.perlbench	C	int	0.9513	1.0000	0.9555	0.9994	0.9964	0.9957 (6)	0.9910	0.9904
401.bzip2	C	int	1.0000	1.0000	0.9877	0.9877	1.0000	1.0000	0.9870	0.9870
403.gcc	C	int	0.9840	1.0000	0.9765	0.9910	0.9812	1.0000	0.9620	0.9825
429.mcf	C	int	1.0000	1.0000	0.9706	0.9706	1.0000	1.0000	0.9706	0.9706
445.gobmk	C	int	0.9725	1.0000	0.9724	0.9992	0.9950	1.0000	0.9948	0.9988
456.hmmer	C	int	0.9956	1.0000	0.9960	0.9980	1.0000	1.0000	0.9937	0.9854
458.sjeng	C	int	1.0000	1.0000	0.9932	0.9932	1.0000	0.9905 (1)	0.9856	0.9786
462.libquantum	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
464.h264ref	C	int	0.9979	1.0000	0.9963	0.9981	0.9976	1.0000	0.9962	0.9962
433.milc	C	fp	0.9946	1.0000	0.9919	0.9959	1.0000	1.0000	0.9959	0.9959
470.lbm	C	fp	1.0000	1.0000	0.9643	0.9643	1.0000	1.0000	0.9630	0.9630
482.sphinx3	C	fp	0.9955	1.0000	0.9941	0.9941	1.0000	1.0000	0.9970	0.9970
471.omnetpp	C++	int	0.9988	1.0000	0.9658	0.9995	0.9929	0.9994 (1)	0.9915	0.9994
473.astar	C++	int	1.0000	1.0000	0.9898	0.9898	1.0000	1.0000	0.9898	0.9898
483.xalancbmk	C++	int	0.9895	1.0000	0.9650	0.9998	0.9801	1.0000	0.9808	0.9999
444.namd	C++	fp	1.0000	1.0000	0.9905	0.9905	0.9787	1.0000	0.9717	0.9904
447.dealII	C++	fp	0.9759	1.0000	0.9786	0.9988	0.9542	1.0000	0.9724	0.9972
450.soplex	C++	fp	0.9972	1.0000	0.9968	0.9989	0.9939	1.0000	0.9946	0.9989
453.povray	C++	fp	0.9692	1.0000	0.9969	0.9994	0.9955	1.0000	0.9955	0.9994
410.bwaves	Fortran	fp	1.0000	1.0000	0.9412	0.9412	-	-	-	-
416.gamess	Fortran	fp	0.9574	1.0000	0.9537	0.9948	-	-	-	-
434.zeusmp	Fortran	fp	0.9063	1.0000	0.9222	0.9651	-	-	-	-
435.gromacs	Fortran	fp	0.9931	1.0000	0.9937	0.9964	-	-	-	-
436.cactusADM	Fortran	fp	0.9985	1.0000	0.9985	0.9992	-	-	-	-
437.leslie3d	Fortran	fp	1.0000	1.0000	0.9375	0.9375	-	-	-	-
454.calculix	Fortran	fp	0.9840	1.0000	0.9851	0.9851	-	-	-	-
459.GemsFDTD	Fortran	fp	1.0000	1.0000	0.9610	0.9487	-	-	-	-
465.tonto	Fortran	fp	0.9901	0.9997 (1)	0.9682	0.9569	-	-	-	-
481.wrf	Fortran	fp	0.9880	1.0000	0.9924	0.9927	-	-	-	-
C average	C	both	0.9909	1.0000	0.9832	0.9910	0.9975	0.9988	0.9864	0.9871
C++ average	C++	both	0.9901	1.0000	0.9791	0.9967	0.9850	0.9998	0.9852	0.9964
Fortran average	Fortran	fp	0.9817	1.0000	0.9653	0.9718	-	-	-	-
Average	all	both	**0.9876**	**1.0000**	**0.9760**	**0.9857**	**0.9929**	**0.9992**	**0.9860**	**0.9905**

Figure 7: SPEC 2006 results

test with SPEC, and have no evaluations on other languages such as C++ or Fortran, only our results are presented.

C++ programs. One thing to note is that our function boundary analysis for C++ programs is a slightly different from C and Fortran programs. This is due to the exception handling feature of the C++ language. Specifically, C++ exception handling results in a stack unwinding operation, followed by a control transfer to exception handler code (also called a "landing pad") in a caller. Since metadata is required to guide this handling, ELF binaries contain such "call frame information" in DWARF format in their `.eh_frame` sections.

We need to properly deal with C++ exception handling because it is a special form of indirect control transfer. If it is not considered, our control flow traversal would not include the landing pads (which actually belong to the function), and our boundary identification would be incorrect. Therefore, we parse the exception handling metadata and the find landing pads to extend the CFG for each function.

Note that exception handling information must be present in every binary, including stripped binaries.

7.4.2 Contributions of Each Step

To understand how each step of analyses contributes to the finally identified functions, we list the corresponding results for SPEC 2006 programs in Figure 8. To conserve space, only GCC (-O2) compiled programs are shown.

We can see that direct calls contribute to the largest number of identified functions. (Note that this includes direct calls made within functions that are only indirectly reached.) The number of direct tail calls varies across programs, but they aren't uncommon in optimized binaries.

"Indirect"(ly) reached functions are largely program dependent. Our system successfully identified a large number of functions of category "indirect" for the programs that make extensive use of indirect calls (e.g., 445.gobmk, 403.gcc), and most of C++ programs that have an abundance of virtual function calls. "Gap" represents our last resort for identifying functions. It uncovers unreachable functions in most cases, but there can be a few false negatives for some programs.

7.4.3 Pruning Spurious Functions

Binary	Total	Direct call	Direct tail call	Indirect	Gap	FALSE neg.
400.perlbench	1742	48.16%	4.59%	37.54%	9.70%	0.00%
401.bzip2	81	60.49%	4.94%	9.88%	24.69%	0.00%
403.gcc	4653	68.41%	4.32%	20.03%	7.01%	0.24%
429.mcf	34	70.59%	5.88%	17.65%	5.88%	0.00%
445.gobmk	2543	26.23%	4.76%	66.50%	2.52%	0.00%
456.hmmer	504	54.76%	5.36%	5.56%	34.33%	0.00%
458.sjeng	146	71.92%	6.16%	8.90%	13.01%	0.00%
462.libquantum	109	67.89%	6.42%	6.42%	19.27%	0.00%
464.h264ref	535	79.44%	4.11%	7.29%	9.16%	0.00%
433.milc	246	79.27%	1.63%	3.25%	15.85%	0.00%
470.lbm	28	71.43%	7.14%	21.43%	0.00%	0.00%
482.sphinx3	338	70.41%	3.25%	3.55%	22.49%	0.30%
471.omnetpp	2036	27.31%	3.78%	56.53%	12.38%	0.00%
473.astar	98	74.49%	2.04%	8.16%	15.31%	0.00%
483.xalancbmk	13525	33.61%	5.02%	52.06%	9.29%	0.01%
444.namd	105	44.76%	1.90%	51.43%	1.90%	0.00%
447.dealII	7242	26.88%	3.87%	30.49%	38.65%	0.11%
450.soplex	935	43.32%	4.28%	39.47%	12.94%	0.00%
453.povray	1639	58.82%	3.60%	28.74%	8.85%	0.00%
410.bwaves	17	52.94%	11.76%	35.29%	0.00%	0.00%
416.gamess	2898	94.76%	1.28%	0.62%	3.31%	0.03%
434.zeusmp	86	65.12%	2.33%	6.98%	23.26%	2.33%
435.gromacs	1100	70.27%	2.73%	3.91%	22.91%	0.18%
436.cactusADM	1311	44.39%	3.13%	14.72%	37.76%	0.00%
437.leslie3d	32	68.75%	9.38%	18.75%	3.13%	0.00%
454.calculix	1338	69.36%	4.26%	0.45%	24.96%	0.97%
459.GemsFDTD	78	76.92%	5.13%	7.69%	8.97%	1.28%
465.tonto	3851	68.84%	6.13%	0.70%	22.51%	1.58%
481.wrf	2888	55.37%	4.95%	0.66%	38.50%	0.52%
Average	1729	**47.99%**	**4.36%**	**29.99%**	**17.42%**	**0.23%**

Figure 8: Functions identified in each step

Binary	Instructions	Candidates	Table jump filtered	Cond jump filtered	Data flow filtered	True positive	False positive
400.perlbench	220844	2269	65.80%	0.40%	1.37%	28.82%	3.44%
401.bzip2	11982	56	85.71%	0.00%	0.00%	14.29%	0.00%
403.gcc	623913	6702	81.86%	0.30%	2.67%	13.91%	1.01%
429.mcf	2557	6	0.00%	0.00%	0.00%	100.00%	0.00%
445.gobmk	162579	2070	11.26%	0.43%	3.09%	81.69%	3.33%
456.hmmer	59038	252	87.30%	0.40%	0.79%	11.11%	0.40%
458.sjeng	22306	142	90.85%	0.00%	0.00%	9.15%	0.00%
462.libquantum	9590	7	0.00%	0.00%	0.00%	100.00%	0.00%
464.h264ref	103065	126	63.49%	1.59%	3.17%	30.95%	0.79%
433.milc	23644	50	76.00%	0.00%	6.00%	16.00%	2.00%
470.lbm	2368	6	0.00%	0.00%	0.00%	100.00%	0.00%
482.sphinx3	34906	19	26.32%	0.00%	5.26%	63.16%	5.26%
471.omnetpp	113430	1249	7.53%	0.00%	0.16%	92.15%	0.16%
473.astar	8802	8	0.00%	0.00%	0.00%	100.00%	0.00%
483.xalancbmk	677697	8232	10.71%	0.39%	1.59%	85.53%	1.43%
444.namd	65546	56	0.00%	1.79%	1.79%	96.43%	0.00%
447.dealII	621668	2712	12.35%	0.26%	2.77%	81.42%	2.80%
450.soplex	86167	569	33.92%	0.00%	0.88%	64.85%	0.35%
453.povray	204876	2019	72.56%	0.20%	1.49%	23.33%	2.23%
410.bwaves	6001	6	0.00%	0.00%	0.00%	100.00%	0.00%
416.gamess	1514821	3099	78.64%	0.84%	15.62%	0.58%	3.94%
434.zeusmp	49940	16	0.00%	0.00%	37.50%	37.50%	25.00%
435.gromacs	192715	394	77.16%	0.25%	9.90%	10.91%	1.27%
436.cactusADM	133661	562	63.88%	0.36%	1.25%	34.34%	0.18%
437.leslie3d	21715	8	0.00%	0.00%	25.00%	75.00%	0.00%
454.calculix	333508	270	74.81%	0.37%	17.78%	2.22%	4.44%
459.GemsFDTD	82660	76	76.32%	0.00%	15.79%	7.89%	0.00%
465.tonto	818471	1652	89.59%	0.54%	6.11%	1.63%	1.45%
481.wrf	785451	587	75.47%	1.36%	17.04%	3.24%	2.56%
Average	241170	1146	**43.50%**	**0.33%**	**6.10%**	**47.80%**	**1.94%**

Figure 9: Effects of spurious function filters

As discussed, function interface verification is critical in pruning spurious functions from the identified candidate set. In this section, we evaluate the effectiveness of each verification mechanism. The results are presented in Figure 9. (Again, only GCC -O2 compiled binaries are shown.)

The number of instructions and the number of candidates (obtained after extracting function-pointer-like constants from the binary) are first shown. Note that for "candidates", the portion of constants that overlap with direct call targets are excluded. It is clear that the candidate set is much smaller than the total number of instructions, from which we can derive potentially indirectly reachable functions. The following three columns present the percentage of candidates that has been ruled out using proposed mechanisms. The final two columns are the percentages of true positives (indirectly reachable functions) and false positives (spurious functions that are not removed).

As shown in Figure 9, compared with other checks, table jump targets filter significantly reduce the number of spurious function starts. This is because, jump table entries are basically code pointers, and will be inevitably included by our start address enumeration. However, they are spurious function starts and should be removed. Table jump target analysis and check represents an effective technique for this task, although it is not strictly required as those targets are likely to be removed by data-flow property verification as well, it is more light-weight than the latter.

Note that our data-flow property verification is able to detect spurious functions that can skip the control-flow checks: as shown in the figure, after control-flow checks, additional spurious functions are pruned. This is because, not all spurious function starts are code pointers such as table jump targets. They could also be a coincidental byte sequence, as discussed in Section 4.2. Moreover, table jumps and their targets may not be fully identified by analysis. Therefore, the numbers for data-flow checks in Figure 9 only represent *additional* capability, not the full potential. When control-flow checks are turned off, the vast majority (if not all) of their captures can be pruned by the data-flow check alternatives.

7.5 Soundness

7.5.1 Soundness for Instrumentation

Since one goal for our function recovery is to enable sound instrumentation, we need to evaluate the results with respect to this goal. Depending on the specific purpose or design, an instrumentation could be sensitive to recall or precision of the recovered functions. We examine both situations next.

Recall-sensitive instrumentation. One prominent example of recall-sensitive instrumentation is coarse-grained CFI [2]. Specifically, consider a CFI policy that an indirect call must target the entry point of one of the legitimate functions in the program. With a recall rate less than 100%, some legitimate function entry points would not have been identified, and hence CFI enforcement based on such an analysis can lead to runtime failures of legitimate programs. Note, however, that an analysis can miss *unreachable* functions without causing problems: since a legitimate program will never target an unreachable function, the enforced CFI property won't break it. For this reason, we show the precision and recall rates achieved for potentially *reachable* functions in Figure 7. We only focus on their starts in this case, because precise function ends are not needed for the instrumentation. We get the ground truth by preserving relocation information during the compilation and linking process, and consider all functions with a relocation

```
080ad0e0 <func1>:
......
80ad0fc:  eb 02 jmp 80ad100 <func2>
80ad0fe:  66 90 xchg %ax,%ax

080ad100 <func2>:
80ad100:  55 push %ebp
80ad101:  57 push %edi
......
```

Figure 10: An unrecognized, tail called function

```
0818ba30 <func>:
818ba30:  flds 0x86ed1d0 // enumerated possible func start
818ba36:  fstpl 0x8ca5af0
......  // other similar instructions
818c784:  fldl 0x87202c0 // enumerated possible func start
818c78a:  fstpl 0x8ca5908
818c790:  fldl 0x87202c8
818c796:  fstpl 0x8ca5910
......  // other similar instructions
818c8d0:  movl $0xf2,0x8ca5a4c
818c8da:  movl $0xf6,0x8ca5aec
818c8e4:  ret
```

Figure 11: A falsely identified (indirectly reachable) function [818c784, 818c8e4]

entry as an indirectly reachable function. Other functions that are directly called by these functions are also added to the reachable set. Finally, our results are evaluated against this set.

As the figure shows, for most programs, we have achieved 100% recall for reachable function starts. For one of the Fortran programs (465.tonto), our analysis missed one reachable function. This function is reached via a jump that our analysis did not correctly identify as a tail call. Indeed, it should be clear from the design of our analysis that it would achieve a perfect recall rate, except possibly for tail calls. The reasoning behind this is as follows: if function pointers don't go through arithmetic or logical operations, then all reachable functions should either be directly called, or their address must be stored somewhere within the binary. It should also be noted that missed tail calls don't cause problems for CFI instrumentation of calls, since functions reachable only via tail calls will never be the target of a call.

Although a high precision rate is not critical for soundness of CFI instrumentation, it impacts security. Therefore, a strategy that sacrifices precision drastically for perfect recall (e.g., the static analyses used by BinCFI) is unattractive. Instead, we achieve perfect recall, while maintaining a high precision of about 97%.

In comparsion, machine learning based systems [6, 27] identify functions with a "matching-known" flavor, and thus represent a *best effort* strategy. Although their high overall recall and precision results are good for many analysis applications, they are less applicable to instrumentation applications that have stringent soundness requirements.

Precision-sensitive instrumentation. Some instrumentations operate on individual functions as a unit [9, 24, 8], and their soundness typically requires that they be applied only to legitimate functions.

Since directly called functions are free of errors and unreachable functions are not relavant, the imprecision could possibly originate from two sources: indirectly reachable functions and (direct) tail called functions.

An address could be incorrectly identified as an (indirectly reachable) function start if our interface verification mechanism was insufficient. Although our comprehensive verification schemes are generally effective and can remove vast majority of the spurious function starts, such misses do happen. Figure 11 shows one example. In this case, since all instructions access global memory and there are no stack or register operations, our interface verification could not identify [818c784, 818c8e4] as a spurious function.

Despite these imprecisions, one distinguishing feature of our system is that the real function which encloses the spurious one is always identified. In Figure 11 for example, [818ba30, 818c8e4] is also recovered (recall in our model, the recovered functions can overlap or share code). And with this property, different measures could be taken for different instrumentations to cope with the imprecisions.

For RAD [9], no work is required at all because the instrumentation is resistant to such imprecisions[5]. For other more complicated instrumentations [24, 8], the overlapping functions could have their own instrumented version (which are disjoint), and an address translation scheme for indirect branches (commonly adopted by binary transformation systems [21, 35, 29]) could be used. With this technique, an indirect call target is translated at runtime to point to its instrumented version before control transfer. Since the falsely identified function is never called at runtime, the incorrect instrumentation will not be executed.

Our system may also falsely recognize intra-procedural jumps as direct tail calls. For precision-sensitive instrumentations, we can employ a simple heuristic to avoid false positives by tightening the tail called function recognition standard. Specifically, we perform a backward scan from the direct jump target. If there are "non-nop" instructions that lead to the target, it is considered as an intra-procedural target and not a function start. Although this overly strict standard may slightly increase false negatives (less than 10 for each SPEC program), the instrumentation soundness is maintained. After incorporating this heuristic, the precision and recall only slightly varied compared with the results in Figure 7.

7.5.2 *Soundness for Analysis*

Although some analyses are quite permissible for function boundary errors, there are other analyses that have stricter requirements. One such example is stack variable detection [5]. The analysis works by summarizing all stack accesses of a function to determine stack variable location and size. The recovered variables could be further used in applications

[5]This is because, at the spurious function start, an extra, unneeded "return address" is pushed to the shadow stack. While this slightly increases attacker's options, it does not break program functionality since at the function epilogue, return addresses is popped repeatedly from the shadow stack until there is a match. Note that the true return address does present in the shadow stack, because the larger, real function is also recovered.

Tool	ByteWeight	NeuralNetwork	Ours
Experiment Setup	desktop quad-core 3.5GHz i7-3770K CPU, 16GB RAM	Amazon EC2 c4.2xlarge instances; each of which: 8-core 2.9GHz Intel Xeon CPU, 15GB RAM	laptop quad-core 1.7GHz i5-4210U CPU, 8GB RAM
Training (10-fold)	275 compute hours (estimate)	20 compute hours	0
Testing	457,997 seconds	1,062 seconds	180,240 seconds

Figure 12: Analysis setting and performance comparison

such as symbolic and vulnerability analysis [3].

These analyses obviously necessitate sound function boundaries. For instance, incorrect results can be produced if a stack-modifying instruction (e.g., `push %reg`) is missed, or an extra one is included — a situation likely to happen for best effort approaches.

Several features of our system help ensure soundness. Since these analyses are more sensitive to precision, a conservative strategy is to simply only consider directly reachable functions, which are already a significant portion (Figure 8). Moreover, even if we take into account indirectly reachable and unreachable functions for better coverage, our interface verification mechanism would prune vast majority of spurious functions, and more importantly, only leaves ones with *limited* error possibility. For instance, the left spurious functions may have no or limited stack and/or register usage (e.g., Figure 11). Therefore, none stack variable is discoverd for this "function" in the analysis, and soundness is not affected. Note that additional, analysis-specific function property checking mechanisms could be easily incorporated to our system for sound analysis.

7.6 Performance

Our focus so far has been on accuracy rather than runtime, and hence we have not made any systematic efforts to optimize performance. Nevertheless, it is useful to compare its performance against previous techniques.

As compared to machine learning based approaches [6, 27], one of the advantages of our approach is that it does not require training. The results of our analysis, together with those from ByteWeight [6] and neural network based system [27], are summarized in Figure 12. The numbers are based on our first dataset.

The neural network based system uses much less time for the testing because it only identifies the bytes where functions start and end, without recovering the function body. As a comparision, ByteWeight and our system follow the CFG to identify function ends, therefore can recognize the exact instructions belonging to the function, and identify physically non-contiguous parts of the function.

Currently, it takes about 3 minutes on average to analyze a binary of our test suite. Although this is already satisfiable for many cases, there are many opportunities for improvement. For example, spurious functions can be immediately spotted if the entry basic block has violating behavior (e.g., modifying return address) and therefore analysis of the whole function can be avoided. This is in contrast with our current naive implementation that complete analysis and checks are performed. Moreover, the number of analysis states could be significantly reduced, by maintaining states for basic blocks instead of instructions.

Calling convention	Stack cleanup by	Argument passing
System V ABI	callee	RDI, RSI, RDX, RCX, R8, R9, then stack
Microsoft x64	callee	RCX, RDX, R8, R9, then stack

Figure 13: Calling convention for x86-64 architecture

8. Discussion

Shared libraries. Other than *executables*, shared libraries represent another common type of binary file. Since prior work [6, 27] focused on executables, we followed the same practice for easier comparison. Nevertheless, our technique works on shared libraries as well. One feature of shared library is actually to our advantage: symbol information for exported functions are preserved, as it is needed for symbol resolution by the dynamic linker. This information is exploited by the first stage of our analysis.

There are only two other differences for shared libraries: first, is the use of position-independent code (PIC), which is a direct call to `thunk` code that retrieves current instruction pointer from stack (the simplest case being `call next`; `next : pop %reg`) and makes direct call target not *definite* function start. Since our data-flow property checking already deals with this (the use of return address), it can be simply applied to direct call targets. The second difference is that "function pointers" and jump table targets are a bit harder to identify, as the constants represent offsets. However, this added complexity has already been addressed in previous work [35], and only a small adaptation to our constant finding method is needed.

x86-64 architecture. Although we focused on x86-32 architecture in the presentation and evaluation, our techniques are general and apply to x86-64 as well. Actually, while the control flow property verification stays the same, x86-64 is advantageous in two ways. First, since the addresses are longer than x86-32 (8 bytes vs 4 bytes), the probability of a constant being misidentified as a code address is reduced. Second, for either Windows platform, or System V, there is only one calling convention available for x86-64 architecture (Figure 13), both of which are more easily violated by spurious functions. This is because, both calling conventions preserve the stack pointer, which simplifies our check. In addition, since there are 16 general purpose registers available, it is very likely that some of the rest 10 (for System V) or 12 (for Windows) registers are used and therefore live at spurious function starts. If those uses are not callee saves, then spurious functions are more readily identified.

Function identification and checking. Our function property checking technique presented in Section 6 is basically a general mechanism for confirming functions. Although in our approach it was applied to tentative functions derived from binary constants, it can be used to test possible functions obtained through other means. For example, possible function starts could be identified using pattern matching or machine learning [6, 27] approaches. Our property checking could then be applied for increased confidence. To reduce false negatives, the procedure could be followed by

a second round of function identification and analysis, and a similar technique that checks the "gap" area could also be employed.

In this work, we used constant scan as one of the function identification mechanism because it is very simple and light weight, and provides a sufficiently precise set for further verification. Our straight-forward directly reachable function identification is also beneficial because for these functions that we already have high confidence, property checking is not needed. However, we point out constant scan and correct disassembly are not strictly needed. Any function start identification (e.g., machine learning) and code discovery/disassembly techniques could be potentially combined. We leave the integration of our function property checking mechanism with other function identification approaches as future work.

9. Related Work

Function (boundary) identification. Many tools recognize functions using call graph traversal and function prologue matching. Examples include the CMU Binary Analysis Platform [7] and the Dyninst instrumentation tool [17]. IDA [1] uses proprietary heuristics and a signature database for function boundary recovery to assist disassembling. Its problem include that it underperforms for different compilers and platforms, and the overhead of maintaining an up-to-date signature database.

Rosenblum et al. first proposed using machine learning for function start identification [23]. The precision and performance has been greatly improved by recent work from Bao et al. [6] and Shin et al. [27], due to adoption of different machine learning techniques such as weighed prefix trees and neural networks. However, as discussed, machine learning based approaches may not be able to produce sound results that can be directly used for instrumentation.

Similar to our work, SecondWrite [12] uses analysis for function recovery and also considers constants identified in binaries as potential function starts. However, they use a different disassembly technique and employ static analysis of code snippets to mark likely spurious code. As a comparison, our interface property verification mechanism is more principled by considering the function interface abstraction/specification, and more comprehensive by taking into account both control-flow and data-flow properties.

Disassembly. The disassembly problem also has some overlap with function boundary identification. Kruegel et al. leverage speculative disassembly to disassemble obfuscated binaries [19]. To make their algorithm scalable, they use pattern based approach to break a binary into functions. Jakstab [18] performs static value-set analysis to resolve indirect control flow targets. However, many indirectly reachable functions are still not identified due to over-approaximation. Wang et al. [33] assume accurate function boundaries to assist their goal of producing disassembled code that can be reassembled.

BinCFI [35] uses linear disassembly with error correction as the first step of instrumenting binaries. However, their work isn't concerned with function boundary identification. In this work, we extend their disassembly technique and leverage static analysis for function boundary recovery, therefore can enable more analysis and instrumentation applications.

Static binary analysis to recover high-level constructs. Because much information has been lost during the compilation process, binaries are difficult to understand or reengineer. Other than function boundaries, previous works also focus on recovering other high-level constructs, such as variables and types [5, 20, 4] or function signatures [13]. The more ambitious goal is to recover source code through decompilation [16, 14, 25]. However, all these are downstream analyses that depend on accurate function boundary information to produce high quality results.

10. Conclusions

In this work, we present a static analysis based approach for function boundary identification in stripped binary code. Compared with previous efforts that rely on matching of code patterns, our approach is more principled by leveraging the function interface abstraction and implementation. To that end, we developed comprehensive checking mechanisms for control-flow and data-flow properties that are associated with the function interface. Evaluations on binaries from three different languages, three compilers and four optimization levels have shown that we can achieve better results than state-of-the-art machine learning based systems. More importantly, we argue that our approach is sound for several common analysis and instrumentation applications.

11. References

[1] Hex rays. https://www.hex-rays.com/index.shtml.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[3] K. Anand, K. Elwazeer, A. Kotha, M. Smithson, R. Barua, and A. Keromytis. An accurate stack memory abstraction and symbolic analysis framework for executables. In *IEEE International Conference on Software Maintenance*, 2013.

[4] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *ACM European Conference on Computer Systems*, 2013.

[5] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you eXecute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2010.

[6] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: Learning to recognize functions in binary code. In *USENIX Security Symposium*, 2014.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *Computer aided verification*, 2011.

[8] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[9] T. Chiueh and M. Prasad. A binary rewriting defense against stack based overflows. In *USENIX ATC*, 2003.

[10] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*. IEEE, 1999.

[11] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security*, 2014.

[12] K. ElWazeer. *Deep Analysis of Binary Code to Recover Program Structure*. PhD thesis, University of Maryland, 2014.

[13] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *ACM PLDI*, 2013.

[14] M. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Working Conference on Reverse Engineering*, 2004.

[15] H. Flake. Structural comparison of executable objects. In *International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment*, 2004.

[16] I. Guilfanov. Decompilers and beyond. *Black Hat USA*, 2008.

[17] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 2005.

[18] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification*, 2008.

[19] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, 2004.

[20] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *NDSS*, 2011.

[21] C. 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 *PLDI*, 2005.

[22] X. Meng and B. P. Miller. Binary code is not easy. In *International Symposium on Software Testing and Analysis*, 2016.

[23] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. Learning to analyze binary computer code. In *AAAI Conference on Artificial Intelligence*, 2008.

[24] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*, 2008.

[25] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Usenix Security*, 2013.

[26] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Working Conference on Reverse Engineering*, 2002.

[27] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*, 2015.

[28] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. (state of) the art of war: Offensive techniques in binary analysis. In *IEEE S&P*, 2016.

[29] 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 *Working Conference on Reverse Engineering*, 2013.

[30] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security. Keynote invited paper.*, 2008.

[31] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *CCS*, 2015.

[32] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE S&P*, 2016.

[33] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *USENIX Security Symposium*, 2015.

[34] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE S&P*, 2013.

[35] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, 2013.