Function Interface Analysis: A Principled Approach for Function Recognition in COTS Binaries

Rui Qiao
Stony Brook University
ruqiao@cs.stonybrook.edu

R. Sekar
Stony Brook University
sekar@cs.stonybrook.edu

***Abstract*—Function recognition is one of the key tasks in binary analysis, instrumentation and reverse engineering. Previous approaches for this problem have relied on matching code patterns commonly observed at the beginning and end of functions. While early efforts relied on compiler idioms and expert-identified patterns, more recent works have systematized the process using machine-learning techniques. In contrast, we develop a novel static analysis based method in this paper. In particular, we combine a low-level technique for enumerating candidate functions with a novel static analysis for determining if these candidates exhibit the properties associated with a function interface. Both control-flow properties (e.g., returning to the location at the stack top at the function entry point) and data-flow properties (e.g., parameter passing via registers and the stack, and the degree of adherence to application-binary interface conventions) are checked. Our approach achieves an F1-score above 99% across a broad range of programs across multiple languages and compilers. More importantly, it achieves a 4x or higher reduction in error rate over best previous results.**

I. 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 transformation techniques require function information. For reverse engineering tasks such as decompiling [21, 18, 33], 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 code search [19, 16, 15, 9], binary code reuse [41], security policy enforcement [12, 32, 10, 2, 42, 46, 43, 39, 40], type inference [26], in-depth binary analysis such as vulnerability detection [38], and more. In fact, a recent survey performed literature study by collecting all binary-based papers published last 3 years at top security conferences, and found that 14 out of 30 works rely on function boundary information [4]. As a result, developers of most existing binary analysis platforms [1, 8, 22, 36] need to design and implement techniques to recognize functions.

Function recognition is a challenging task for stripped COTS binaries since they lack debug, relocation, or symbol information. Unlike source code, functions in binaries can have multiple entry points, potentially causing multiple functions to be recognized in the place of one. Moreover, while directly called functions can be easily identified after disassembly, there exist a significant fraction of functions that are reachable only indirectly, i.e., their entry points are known only at runtime. Existing techniques (e.g., [45]) for determining indirectly reached functions tend to highly over-estimate their number, thus leading to poor precision.

Compiler optimizations further exacerbate function recognition in COTS binaries. For instance, *tail call optimization* results in functions being entered via a `jump` instruction instead of a `call` instruction. In addtion, function inlining may eliminate all calls to a function, thereby resulting in *unreachable* functions. 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. Our evaluation shows that the percentage of unreachable functions is significant: over 15% on average for SPEC 2006 programs.

Due to the above difficulties, one cannot rely on the obvious approach of identifying functions by following direct calls. Many previous systems [1, 8, 22, 36] relied instead on pattern-matching function prologues (e.g., the instruction sequence `push ebp`; `mov esp, ebp`) and epilogues. Unfortunately, this approach is far from robust, since these patterns may differ across compilers. Moreover, optimizations may split and/or reorder these code sequences. Other optimizations (e.g., reuse of `ebp` as a general purpose register) may also remove such identifiable prologues/epilogues. As a result, the best existing tools are still unsatisfactory for function recognition [7].

To overcome the limitations of manually identified patterns, machine-learning based approaches have been proposed for function recognition [31, 7, 35]. The idea is to use a set of binaries to train a model for recognizing function starts and ends. Machine learning can build more complete models that work across multiple compilers, while reducing manual effort. As a result, ByteWeight [7] achieved an average F1-score of 92.7% on a benchmark consisting of x86 binaries. Shin et al [35] further improved the accuracy to achieve an F1-score of 94.4% on the same dataset. Unfortunately, error rates of over 5% are still too high for most applications. More importantly, the accuracy of these techniques can be skewed by the choice of the training data. In fact, an independent evaluation of this dataset [5] found many functions to be duplicated across the training and testing sets, thus artificially increasing their F1-score. When evaluated with a different data set, ByteWeight's accuracy degraded to around 60% [5].

In light of these drawbacks of machine-learning based

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

approaches, we propose a more conventional approach for function discovery, one that is founded on static analysis. However, unlike previous techniques that relied on simple control-flow analyses, and were confounded by the above-mentioned complications posed by stripped COTS binaries, our technique incorporates two key advances:

- We develop a *fine-grained analysis* that is based on detailed semantics of every instruction[1], including their effect on the contents of the registers and memory. As a result, our analysis can reason about the content of the stack, as well as the flow of data between a function and its caller.
- We identify a rich set of *data-flow* properties that characterize function interfaces, such as the use of registers and the stack to pass parameters and/or return values. We present a static analysis to discover these flows, and verify whether a candidate function satisifies these properties.

As a result of these advances *we have achieved a 4-fold reduction* in error rate as compared to the results reported by Shin et al [35]. As compared to Nucleus [5], which relies on a static analysis of control-flows, we achieve an even more impressive error rate decrease of more than 7x.

I-A. Contributions

We develop a novel, static analysis based approach for function recognition in COTS binaries. Specifically, we make the following contributions:

- *Function identification by checking function interface properties*. We show that function *interface* properties, as compared to function prologue patterns, can provide valuable evidence for function recognition. We identify a collection of such properties and present static analysis techniques to check them. Each of these techniques is shown to be independently effective in our evaluation.
- *In-depth evaluation*. Our evaluation consists of about 2400 binaries resulting from 312 distinct C, C++ and Fortran programs. These binaries have been compiled using 3 different compilers (GCC, LLVM and Intel) for two architectures (x86 and x86-64) at four distinct optimization levels. In contrast with previous work, our evaluation set includes low-level code with hand-written assembly code, in particular, GNU libc.
- *Highly accuracy*. Our approach achieved an average F1-score of 99% across these data sets, much better than the 90% to 95% achieved by previous works [7, 35, 5]. This represents a reduction in error rate by more than 4x.
- *Deeper insight*. Our approach automatically categorizes recognized functions by their reachability such as "tail-called" or "unreachable." As discussed in Section VIII, such information can be the basis for further tuning and refinement of the analysis in order to support demanding applications such as binary instrumentation that cannot tolerate errors.

[1]We have previously shown how to extract such semantics using existing compilers for a wide range of instruction sets [25, 24, 23].

II. Background

II-A. Organization of Binaries

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 normally stripped off before COTS binaries are distributed.

II-B. Disassembly

Disassembly is usually the first step for any binary analysis. There are two major techniques for disassembly: linear sweep and recursive traversal [34]. 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 [4] and error correction mechanisms [45]. 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. Robust disassembly has been achieved by these techniques for a wide range of complicated and low-level binaries [45, 4], so we rely on the same techniques in this paper.

II-C. Discovering Possible Code Pointers

Although it is undecidable whether a constant value in a binary represents a code pointer, conservative analysis techniques have been developed that identify a superset of possible code pointers. One recent approach [45] is to scan all constants in the binary, and select the subset that (a) fall within the range of code sections within the binary, and (b) target a valid instruction boundary. We start from this conservative set, and develop techniques that prune away almost all non-functions. As shown in our experiments, our analysis reduces the number of valid function pointers by a factor of 3.

II-D. Jump Table Analysis

A jump table is an array of addresses that are possible targets for an indirect jump (which we refer to as a *table jump*). Jump tables are generally used to implement intra-procedural `switch-case` statements in high-level languages. Existing work has developed analysis techniques to identify jump tables as well as their targets [13, 28]. The basic idea is to perform a backwards program slicing from each indirect jump instruction, and then compute an expression for the jump

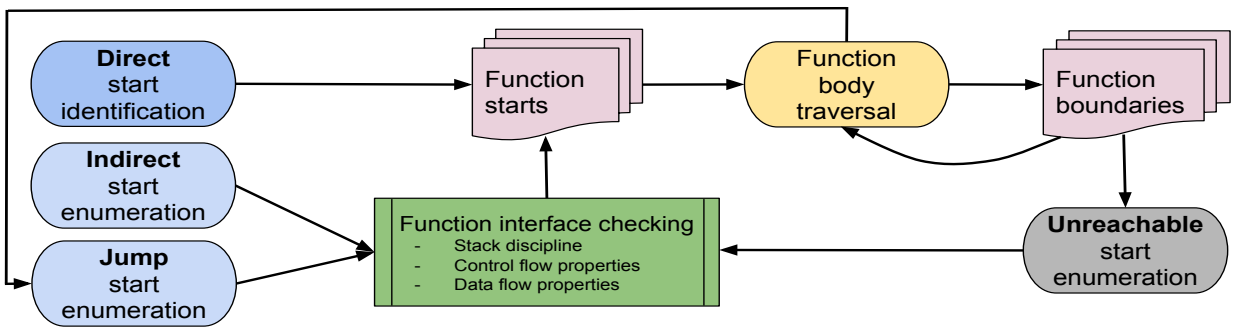

Fig. 1. Overview of our analysis. `Direct` function starts are identified from call instructions in the disassembly, and require no further confirmation. The remaining function start candidates (`Indirect`, `Jump` and `Unreachable` function) need to pass our function interface checks that eliminate spurious functions. Function body traversal is used to determine function ends, and takes advantage of already identified functions. Function body traversal and boundary information feeds back into the determination of unreachable functions, as well as jump-reached (i.e., tail-called) functions.

target. If the expression matches commonly used table jump patterns, the indirect jump is recognized as a table jump. The address of jump table can be extracted from the same expression, and its bound is obtained based on constraints imposed on the index variable. Finally, jump targets can be collected from the identified jump table.

III. Overview of Approach

III-A. Problem Definition

We define a *binary function* as a *contiguous* byte sequence in code section that has one or more *entry points* reached from outside the function; and one or more *exit points* that transfer control from the function to some code outside. Note that the entry point is typically reached using *call* instructions, but other instructions are possible as well, e.g., a *jump*. An exit occurs typically via a *return* instruction, but there are exceptions such as *jump* instructions or calls to non-returning functions such as *exit*.

Our task is to recover bytes belonging to each function. Similar to prior work [7, 35], correctness is determined by matching the *start* and *end* address for each function with symbol table information[2]. Note that the start and end addresses correspond to the smallest and largest addresses among the *bytes* in the function.

Scope. Our analysis focuses on stripped COTS binaries: no debug or symbol information is available. Our evaluation focuses on Executable and Linkable Format (ELF) binaries from x86-32 and x86-64 Linux platform, although our technique itself is applicable to other platforms and binary formats, such as Windows and the Portable Executable (PE) format. We make no assumptions on the source language, compiler used, compiler switches or optimization levels. However, similar to prior work [7, 35, 5], obfuscated binaries are out of scope.

[2]While our experiments are performed on stripped binaries, we rely on symbol tables in unstripped binaries for the ground truth.

III-B. Approach Overview

As illustrated in Fig. 1, our approach involves *enumerating* possible function starts, and then using a static analysis to *confirm* them. Possible function start addresses are enumerated in different ways. Directly called function starts are readily obtained from disassembled code. For indirectly reachable functions, code addresses buried in all binary sections serve as function start candidates, while for unreachable functions, the beginning of unclaimed code regions are considered.

As shown in Fig. 1, any function that isn't directly reached needs to be confirmed through *interfaces*, our approach identifies *spurious* functions by checking for properties associated with function interfaces, such as the stack discipline, and the expected control-flow properties and data-flow properties.

To determine function ends, function body traversal is performed. Tail calls are also identified during this traversal.

In a nutshell, our approach *iteratively* uncovers functions based on how they are reached. Directly reached functions are first identified, and then indirect functions are enumerated and checked. Finally, unreachable functions are handled. Note that `Jump` function enumeration and checking happens alongside the body traversal for all other functions. The whole procedure ends when all code regions have been covered.

Note that multiple-entry functions are supported by our approach: a function with n entry points is treated as if there are n independent single-entry functions. Each is analyzed independently by our method. Multi-entry functions can be easily derived if needed by the applications.

In the following sections, we describe our techniques for determining function starts (Section IV), function boundaries (Section V), and interface checking (Section VI).

IV. Function Starts

IV-A. Directly Reachable Functions

According to our definition in Section III-A, functions are code sequences that are *called* (or alternatively, reached using *jumps*). Therefore, with the disassembly obtained (Section II-B), the targets for *direct* call instructions are *definite*

function starts. They are first collected. Note that we exclude call instructions used for "non-call" purposes [30], such as retrieving current instruction pointer in the case of position independent code.

Although we can obtain direct jump targets in the same manner, it is non-trivial to distinguish whether they are function starts (as in the case of a tail call), or, more likely, intra-procedural targets. We enumerate jump targets as possible function starts if the target is physically non-contiguous with current function body (Section V). An analysis of the *jump context* is later performed to confirm the function start.

IV-B. Indirectly Reachable Functions

Some functions are only reachable *indirectly.* These include functions that are reached using either indirect *calls*, or indirect *jumps* (i.e., indirect tail calls). To enumerate their starts, constant scanning described in Section II-C is used. Since spurious function starts may also be included, the constants need to be confirmed using interface checking.

IV-C. Unreachable Functions

Unreachable functions are identified by analyzing "gap" areas that do not belong to any of the functions identified by the techniques described so far. Since functions typically have padding bytes after their end, we consider the first non-NOP instruction[3] in each gap as a potential function start. The corresponding function end is then determined using techniques described in Section V. 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, 22, 8], the primary difference is that the identified functions have to pass interface property checking.

V. Identifying Function Boundaries

To identify function boundaries, we traverse a function body, starting at its entry point. All possible paths are followed until control flow exits the function. The largest address of any instruction discovered using this process is considered the end of the function. Note that exits may sometimes take place via jumps (tail calls), or calls to non-returning functions. As described below, we discover and handle those cases as well.

Function body traversal works by following all intra-procedural branches until function exits. Specially, for conditional jumps, both branches are taken, while for table jumps, all recovered targets are followed.

For function body traversal, some special control flow transfers need to be taken into account, most notably C++ exception handling. When an exception occurs, either at the current function or some of its callees, control is first directed to C++ runtime, which is responsible for locating the proper handler code (also called a "landing pad"), and during this process, stack unwinding may be performed. If current function is identified to have a landing pad designated for the raised exception, control flow is transferred to this landing pad. Since a landing pad is essentially *indirectly* reached from C++ runtime, the control flow transfer is not captured in the disassembly of the analyzed binary. We therefore parse the "call frame information" available in `.eh_frame` sections of ELF binaries, the same metadata used by the C++ runtime to guide exception handling. Note that exception handling information must be present even in stripped binaries.

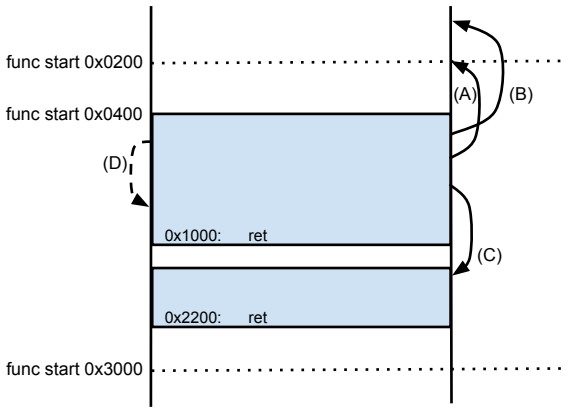

Fig. 2. Tail call detection

Function body traversal stops at function exits. While most functions exit via return instructions, there are special cases that involve calls and jumps as described further below.

Function exits via calls to non-returning functions. Although most functions do return to the caller, some don't. For example, libc `exit` function terminates the program, and the control flow never returns back to the caller. Such calls should be recognized as function exits.

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

Function exits using jump instructions. Tail call is another special type of function exit that uses *jump* instructions. If not recognized, tail calls will be treated as normal intra-procedural jumps, leading to errors in function end identification.

We utilize previously identified functions to detect tail calls. In particular, a jump is classified as a tail call if:

- the jump target is a known function start or a PLT[4] entry;
- or, if the edge crosses known function boundaries.

We illustrate this process with an example. In Fig. 2, function body traversal begins from an identified function start address

[3]We consider an instruction as "NOP" based on its semantics: i.e., the machine state (other than program counter) is *not* changed. For example, other than `nop` itself, `xchg ax, ax` is also a NOP instruction.

[4]PLT (procedure linkage table) entries are code stubs in ELF binaries used to route calls to functions in another binary module (e.g., a shared library).

0x0400. Let A, B, C, and D be four direct jumps encountered during this traversal. Since jump A targets a previously identified function and B crosses a function start, they are identified as tail calls. For jumps C and D we use the following two-step process to determine if they are tail calls. First, our function body traversal *speculatively* follows all jumps and only terminates at definite function exits (e.g., returns). Any jump such as C whose target isn't physically contiguous with other traversal-covered instructions is identified as a potential tail call. In Fig. 2, the colored area starting at 0x400 represents the instructions covered by such a traversal. Since D targets an instruction in this region, it is classified as intraprocedural.

A candidate such as C is confirmed as a tail call if both the jump and the target pass function interface checks. Function *entry* checks are described in the next section. For function *exit*s, our checks relate to the stack discipline. Specifically, the stack pointer value at the jump shouldn't be lower than its initial value on function entry. Otherwise, local storage allocated in the function wouldn't have been freed, suggesting an intra-procedural jump.

In addition to determining function ends, tail call identification serves a second important purpose: enumerating function starts that are reached only through jumps.

VI. Interface Property Checking

For potential functions reached only via jumps or indirect calls, we identify a set of checks to determine if the targets are indeed functions. These checks can relate to control-flow or data-flow properties, as described further below.

VI-A. Control Flow Properties

These properties are designed to identify control transfers into (or out of) a function body that do not conform to a function interface. A function candidate is flagged as *spurious* when such non-conformant control transfers are discovered.

Function entries. We rule out known intra-procedural control transfer targets from being function starts. In particular, table jumps are intra-procedural control flow transfers as they result from `switch-case` statements. Hence, table jump targets are ruled out as possible function starts. Since our jump table analysis is designed to identify *definite* table jump targets, it is safe to remove them from the list of possible function starts.

Function exits. Our second control flow check examines each return instruction to determine if it will transfer control to the address stored on top of the stack at the function entry point.

In most cases, our static analysis can accurately reason about stack pointer value at function exits. Even functions that change their stack pointer by an unbounded amount (e.g., due to `alloca`) typically pass the check: they contain an instruction to move `esp` to `ebp` near their start, and another to move `ebp` back to `esp` near their end. Our static analysis can hence conclude that the return address will be read from the same location where it would have been stored by the call instruction used to enter the function. Coupled with an analysis of stores on stack, it is possible to determine if this location was preserved during function execution.

```
805ce70 <get_date>:
805ce70:     push  %ebp  ; real func start
805ce71:     push  %edi
805ce72:     push  %esi
805ce73:     push  %ebx
...          ...
805d900:     pop   %ebx  ; +4 ; spurious func start
805d901:     pop   %esi  ; +8
805d902:     pop   %edi  ; +12
805d903:     pop   %ebp  ; +16
805d904:     ret
```

Fig. 3. Incorrect return address is used for a spurious function

```
8081130 <find_connection_moves>:
...          ...
8081136:     sub   $0x533c,%esp
...          ...
8081a10:     mov   %edi,0x4(%esp) ; spurious func start
8081a14:     mov   $0x80e6718,(%esp)
8081a1b:     call  807c8d0
...          ...
```

Fig. 4. "Return address" is overwritten for a spurious function

The main purpose of function exit check is to flag spurious functions, i.e., our focus is on detecting violations of this property. To illustrate its use, consider the code snippet in Fig. 3. In this snippet, 0x805ce70 is a real function start, while 0x805d900 isn't. They are both enumerated as potential function starts. However, function [0x805d900, 0x805d904] is detected as spurious by our analysis, as its return address comes from a location 16 bytes above the correct stack slot.

Fig. 4 shows our second example. In this code, address 0x8081a10 is enumerated as a potential function start. However, since its "return address" is overwritten at 0x801a14, the "function" can never return to the intended return address. Hence 0x8081a10 is flagged as spurious function start. Note that in the context of the real function starting at 0x8081130, the instruction at 0x801a14 does not modify the return slot, so 0x8081130 isn't flagged spurious.

Internal Instructions. A function's internal instructions should not be targeted by control flows from outside. An exception arises in the case of multi-entry functions, but even then, these alternate entry points must be targeted by *inter-procedural* control transfers. In contrast, if an internal instruction of a function f is targeted by an *intra-procedural* transfer from another function g, that provides strong evidence that f is likely spurious.

Recall that when we perform interface verification for a function beginning at location f, we start with a traversal of its body at f. The instructions uncovered by this traversal constitute the body of f, and any control transfers using intra-procedural control flow constructs (e.g., table jumps) from outside this body indicate that f is spurious.

Registers	x86-32 Windows	x86-32 UNIX-like	x86-64 Windows	x86-64 UNIX-like
Allowed argument	(See Fig. 6	(See Fig. 6)	`rcx`, `rdx`, `r8`, `r9`	`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
Callee-save	`ebx`, `esi`, `edi`, `ebp`	`ebx`, `esi`, `edi`, `ebp`	`rbx`, `rsi`, `rdi`, `rbp`, `r12-r15`	`rbx`, `rbp` `r12-r15`

Fig. 5. Register usage summary for calling conventions on different platforms

x86-32 calling convention	Argument passing
cdecl, stdcall, pascal	stack
fastcall (Microsoft, GNU)	`ecx`, `edx` then stack
fastcall (Borland)	`eax`, `edx`, `ecx` then stack
thiscall (Microsoft)	`ecx` then stack

Fig. 6. Arguments passing for x86-32 calling conventions

VI-B. Data Flow Properties

Function calling conventions [20] govern the flow of data between callers and callees of legitimate functions; in contrast, spurious functions are likely to violate these conventions. While calling convention checks could be applied to dataflows that take place via registers as well as the stack, our implementation only targets register-based flows.

Calling conventions on the most common platforms are summarized in Fig. 5. In this figure, allowed argument registers are registers that can be used for passing arguments to a function, therefore used for inwards data flow. On the other hand, callee-save registers are those registers whose values need to be saved before being used in a function, and restored before the function returns.

Note that on x86-32 allowed argument registers are calling convention specific, and the details are presented in Fig. 6. To compute a reference set, we take the union of all calling conventions. Therefore, the resultant allowed argument registers are `eax`, `edx`, and `ecx`. Any dataflow from a caller to a callee via other registers would be in violation of all calling conventions, thus indicating a spurious function.

Static Analysis for Argument Registers.

If a register is *live* at a "function" start, it is potentially an argument register. This is because a live register indicates its *use* is before its *definition* in the "function" body. Consequently, it must have been defined before the function being called 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 adopts a simple strategy by not considering register saves on the stack as a use of the register. Specifically, if a register is saved to an address less than the value of the stack pointer at function entry, then it is *not* treated as a use of that register.

Note that a special case for argument register checking is that `EFLAGS` are not used for passing information. Therefore, a live flag also suggests a spurious function.

A concrete example for argument register checking is shown in Fig. 4. For the entry point `0x8081a10`, a non-permitted

```
805ce70 <get_date>:
805ce70:     push   %ebp  ; real func start
805ce71:     push   %edi
805ce72:     push   %esi
805ce73:     push   %ebx
...          ...
805d900:     pop    %ebx  ; +4 ; spurious func start
805d901:     pop    %esi  ; +8
805d902:     pop    %edi  ; +12
805d903:     pop    %ebp  ; +16
805d904:     ret
```

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

Fig. 7. The analysis states of example code

argument register (`edi`) is live, thus flagging it as spurious.

Static Analysis for Callee-saves. To check value preservation for callee-save registers, we keep track of register and memory values by performing an abstract interpretation [14]. Our abstract domain is similar to the one described by Saxena et al [32]. In short, each domain element is a sum of a symbolic base which denotes the original register value on 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.

Fig. 7 shows the initial and end states from our analysis of an example snippet. 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 registers `ebx`, `esi`, `edi`, `ebp` do not preserve their values but instead get new values from "return address" (location [ESP + 0]) and "stack argument" region (location [ESP + 4] to [ESP + 12]) , it is recognized as spurious. On the other hand, "function" [0x805ce70, 0x805d904] passes callee-save register usage checking. Note that in case register values end up with `TOP`, our analysis conservatively concludes no violations.

Note that in above two examples (Fig. 3 and Fig. 4), spurious functions violate *both* control flow and data flow properties. This is not always the case — sometimes only a single interface checking technique is effective. Thus, by combining these checks, we significantly decrease the odds of spurious functions passing all checks.

VII. Evaluation

VII-A. Data Set

We used three data sets for evaluating our system.

Data Set 1. The first data set is the same as that used by machine learning based approaches, namely, ByteWeight [7]

	Function start							Function boundary						
	x86-32			x86-64			**overall**	x86-32			x86-64			**overall**
Tool	P	R	F1	P	R	F1	**F1**	P	R	F1	P	R	F1	**F1**
ByteWeight	0.9841	0.9794	0.9817	0.9914	0.9847	0.9880	**0.9849**	0.9278	0.9229	0.9253	0.9322	0.9252	0.9287	**0.9270**
Neural	0.9956	0.9906	0.9931	0.9880	0.9780	0.9830	**0.9881**	0.9775	0.9534	0.9653	0.9485	0.8991	0.9232	**0.9443**
Ours	0.9978	0.9920	0.9948	0.9960	0.9948	0.9954	**0.9951**	0.9865	0.9809	0.9837	0.9912	0.9900	0.9906	**0.9871**
Error ratio	**2.0000**	**1.1750**	**1.3269**	**2.1500**	**2.9423**	**2.6086**	**2.4286**	**1.6667**	**2.4398**	**2.1288**	**5.8523**	**7.4800**	**7.5851**	**4.3178**

Fig. 8. Function start and boundary identification results from different tools

and the work of Shin et al [35]. Since our current implementation is limited to Linux ELF binaries, the comparison focuses on the subset of the results for this platform. Note that the vast majority (2064 of 2200) of the binaries in this data set were on Linux, so our results do cover over 90% of the data used in these works. These 2064 binaries correspond to `binutils`, `coreutils` and `findutils`. They are compiled with GNU (`gcc`) or Intel (`icc`) compilers, from no optimization to the highest optimization level for both x86-32 and x86-64 architectures. These binaries include around 600,000 functions in total, with a total size over 280MB.

Despite its size, this data set is found to be skewed by a recent work [5]. Specifically, since many binaries are from the same package, they share a significant number of functions. This gives machine learning techniques advantages because functions used for learning and testing significantly overlap. Nevertheless, we use this data set in order to provide a direct comparison with machine learning approaches.

Data Set 2. Our second data set is the set of SPEC 2006 programs. As compared to the first data set, 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). Moreover, unlike the first data set, SPEC programs rarely share functions with each other. To compile these programs, we used the GCC compiler suite (`gcc`, `g++`, and `gfortran`) and LLVM (`clang`, `clang++`), and compiled with all optimization levels (O0-O3).

Data Set 3. Our third data set is the GNU C library, which is a suite of functionally related shared libraries (24 in total), including `libc.so`, `libm.so`, `libpthread.so`, etc. This is a more challenging data set for two reasons. First, the code is low-level and contains many instances of hand-written assembly. Second, the binaries are in the form of position independent code (PIC). We used GCC -O2 to compile for both x86-32 and x86-64 architectures.

VII-B. Metrics

We use the same metrics, *precision*, *recall*, and *F1-score* as in previous work [7, 35]. 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:

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

VII-C. Implementation

Our main analysis framework is implemented in Python, and consists of about 4100 lines of code. For the disassembler, we used `objdump` and reimplemented the error correction algorithm from BinCFI [45]. The main framework also includes all major components described, including function start identification, function body traversal, and part of interface checking. Our current analysis engine is based on angr [36].

We used angr mainly because it is a comprehensive binary analysis platform, and supports both x86-32 and x86-64. We built our customized analysis on top of angr, but not using any of its built-in function recovery algorithms. In fact, their accuracy is well under the best published results from machine learning systems [7, 35] and Nucleus [5] (which we compare in the following sections), probably because the primary goal of angr is for offensive binary analysis [36], rather than robust recovery of program constructs in benign binaries.

VII-D. Summary of Results

Fig. 8 summarizes function start and boundary identification results for the first data set. Since for this data set the two machine learning works [7, 35] have best published results and outperformed previous tools (such as IDA and Dyninst

Dataset & compiler	Tool	x86-32 binaries			x86-64 binaries		
		P	R	F1	P	R	F1
SPEC (GCC)	Nucleus	0.97	0.89	0.92	0.97	0.90	0.93
	Ours	0.9988	0.9869	0.9927	0.9952	0.9861	0.9905
SPEC (LLVM)	Nucleus	0.95	0.88	0.91	0.94	0.86	0.90
	Ours	0.9978	0.9933	0.9955	0.9934	0.9902	0.9918
GLIBC (GCC)	Nucleus	-	-	-	-	-	-
	Ours	0.9846	0.9914	0.9879	0.9804	0.9840	0.9817

Fig. 9. Function boundary identification results for SPEC 2006 and GLIBC.

Program	Language	Suite	GCC						LLVM					
			x86-32			x86-64			x86-32			x86-64		
			P	R	F1	P	R	F1	P	R	F1	P	R	F1
400.perlben.	C	int	0.9971	0.9983	0.9977	0.9743	0.9954	0.9847	0.9952	0.9940	0.9945	0.9898	0.9886	0.9892
401.bzip2	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.000	1.0000	1.0000	1.0000	0.9867	0.9736	0.9801
403.gcc	C	int	0.9959	0.9893	0.9926	0.9935	0.9946	0.9941	0.9977	0.9982	0.9979	0.9947	0.9942	0.9944
429.mcf	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9143	0.9697	0.9411
445.gobmk	C	int	0.9996	0.9996	0.9996	0.9980	0.9996	0.9988	0.9996	1.0000	0.9998	0.9996	0.9992	0.9994
456.hmmer	C	int	1.0000	0.9980	0.9990	0.9941	1.0000	0.9970	1.0000	1.0000	1.0000	0.9958	0.9916	0.9937
458.sjeng	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
462.libquan.	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
464.h264ref	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9981	0.9991	1.0000	0.9981	0.9990
433.milc	C	fp	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
470.lbm	C	fp	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
482.sphinx3	C	fp	1.0000	1.0000	1.0000	0.9824	0.9911	0.9867	1.0000	1.0000	1.0000	0.9819	0.9939	0.9879
471.omnet.	C++	int	0.9990	0.9946	0.9968	0.9975	0.9853	0.9914	0.9974	0.9604	0.9786	0.9963	0.9564	0.9759
473.astar	C++	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
483.xalan.	C++	int	0.9911	0.9923	0.9916	0.9883	0.9911	0.9897	0.9958	0.9888	0.9922	0.9920	0.9886	0.9903
444.namd	C++	fp	1.0000	1.0000	1.0000	1.0000	0.9904	0.9951	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
447.dealII	C++	fp	0.9601	0.9577	0.9592	0.9698	0.9494	0.9595	0.9942	0.9707	0.9823	0.9889	0.9731	0.9810
450.soplex	C++	fp	1.0000	0.9764	0.9881	1.0000	0.9443	0.9713	1.0000	0.9847	0.9923	1.0000	0.9858	0.9929
453.povray	C++	fp	0.9974	0.9707	0.9839	0.9913	0.9696	0.9802	1.0000	0.9566	0.9778	1.0000	0.9573	0.9782
C overall	C	both	0.9977	0.9950	0.9963	0.9918	0.9966	0.9942	0.9982	0.9982	0.9982	0.9949	0.9946	0.9948
C++ overall	C++	both	0.9839	0.9808	0.9823	0.9845	0.9757	0.9801	0.9959	0.9792	0.9875	0.9922	0.9796	0.9859
F overall	F	fp	0.9902	0.9846	0.9874	0.9866	0.9826	0.9846	-	-	-	-	-	-
Overall	all	both	**0.9886**	**0.9849**	**0.9868**	**0.9852**	**0.9781**	**0.9817**	**0.9965**	**0.9847**	**0.9906**	**0.9930**	**0.9840**	**0.9885**

Fig. 10. SPEC 2006 results: C/C++ programs and overall

Program	Language	Suite	GCC					
			x86-32			x86-64		
			P	R	F1	P	R	F1
410.bwaves	F	fp	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
416.gamess	F	fp	0.9931	0.9951	0.9941	0.9931	0.9979	0.9955
434.zeusmp	F	fp	1.0000	0.9884	0.9941	0.9767	0.9882	0.9824
435.gromacs	F	fp	0.9991	0.9981	0.9986	0.9982	0.9982	0.9982
436.cactus.	F	fp	1.0000	1.0000	1.0000	1.0000	0.9977	0.9988
437.leslie3d	F	fp	1.0000	1.0000	1.0000	0.9667	0.9355	0.9509
454.calculix	F	fp	0.9940	0.9836	0.9887	0.9953	0.9514	0.9728
459.Gems.	F	fp	0.9737	0.9487	0.9610	0.9733	0.9481	0.9605
465.tonto	F	fp	0.9760	0.9598	0.9678	0.9634	0.9634	0.9634
481.wrf	F	fp	0.9961	0.9958	0.9960	0.9972	0.9955	0.9963
F overall	F	fp	0.9902	0.9846	0.9874	0.9866	0.9826	0.9846

Fig. 11. SPEC 2006 results: Fortran programs

[22]), we only compare our results with them. Because we tested with the same data set, we directly use the numbers reported by them. In this figure (and following ones), "P" denotes precision, while "R" denotes recall. Note that for each architecture, every number (for P/R/F1) is a mean over the corresponding 1032 binaries. To enable direct comparison with machine learning systems, we followed their practice by using an arithmetic mean[5]. The "error ratio" in the figure is defined as $(1 - MAX(ByteWeight, Neural))/(1 - Ours)$ for each column. Our technique reduces the error rate for boundary identification by a factor of 4.32, thus representing a major improvement in accuracy.

The results for our second and third data sets are presented in Fig. 9. We compare with the most recent work in this field, Nucleus [5], which is also based on static analysis. We followed their way of summarizing results: using an average of geometric means for all optimization levels. Our F1 scores for this data set are consistently above 0.99, significantly higher than those of Nucleus (around 0.92). We omitted zooming into each optimization level since our results are not sensitive to them: the F1 score differences are within 0.01.

As shown in the last two lines of the figure, *ours is the first work that evaluates with GLIBC*. Despite the challenges posed by PIC-code, hand-written assembly and other low-level features, our techniques achieve an F1-score above 0.98.

VII-E. Detailed Evaluation

In this section, we present detailed evaluation for our second data set: SPEC 2006 programs. For space reasons, we focus on the most widely used optimization level: -O2. As shown in Fig. 10 and Fig. 11, for a wide range of applications written in three different languages (C, C++ and Fortran) and compiled with two compilers (GCC and LLVM[6]), our overall F1-scores are no lower than 0.9817 for function boundaries. Note that overall metrics are computed by using the aggregated true positives, false positives and false negatives over the selected fraction of binaries. For many individual binaries, we have achieved perfect (1.0000) precision and recall.

Distribution of Different Call Types. To understand how each step of analyses contributes to the finally identified functions, we list the corresponding results for SPEC 2006 programs in Fig. 12. To conserve space, only GCC (-O2) compiled programs for x86-32 architecture are shown. Note that x86-64 binaries have similar results to their x86-32 counterparts.

[5]When using geometric or harmonic mean to summarize our results, the difference is less than 0.0006.

[6]LLVM does not have an official frontend for Fortran.

Binary	Total funcs.	Direct call (%)	Direct jump (%)	Indirect (%)	Unreachable (%)
400.perlben.	1742	48.22	0.46	40.18	11.14
401.bzip2	81	61.73	1.23	9.88	27.16
403.gcc	4653	68.56	2.82	20.89	7.57
429.mcf	34	73.53	0.00	17.65	8.82
445.gobmk	2543	26.27	0.55	70.31	2.87
456.hmmer	504	54.96	2.98	5.56	36.31
458.sjeng	146	72.60	4.11	8.90	14.38
462.libquan.	109	68.81	3.67	6.42	21.10
464.h264ref	535	79.63	2.80	7.29	10.28
433.milc	246	79.67	0.81	3.25	16.26
470.lbm	28	75.00	0.00	21.43	3.57
482.sphinx	338	70.71	1.18	3.85	24.26
471.omnet.	2036	27.36	1.82	56.93	13.36
473.astar	98	75.51	0.00	8.16	16.33
483.Xalan	13525	33.62	2.60	53.84	9.18
444.namd	105	45.71	0.00	51.43	2.86
447.dealII	7242	26.90	1.20	30.46	37.21
450.soplex	935	43.42	3.32	38.93	11.98
453.povray	1639	58.88	1.59	30.14	6.47
410.bwaves	17	58.82	0.00	35.29	5.88
416.gamess	2898	94.79	1.04	0.59	3.49
434.zeusmp	86	66.28	0.00	6.98	25.58
435.gromacs	1100	70.36	1.09	4.09	24.36
436.cactus.	1311	44.47	0.76	14.80	39.97
437.leslie3d	32	71.88	0.00	18.75	9.38
454.calculix	1338	69.43	2.24	0.45	26.83
459.Gems.	78	78.21	2.56	6.41	10.26
465.tonto	3851	68.87	4.05	0.73	24.51
481.wrf	2888	55.40	3.84	0.66	39.89
Overall	**50138**	**48.06**	**2.16**	**30.83**	**17.70**

Fig. 12. Functions identified in each step

Binary	Total pruned	Control flow checking (%)			Data flow checking (%)	
		entry	exit	internal	argument	callee-save
400.perlben.	1630	91.60	79.57	39.26	57.67	53.31
401.bzip2	48	100.00	33.33	89.58	85.42	87.50
403.gcc	5845	94.06	86.35	38.25	71.87	46.14
429.mcf	0	0.00	0.00	0.00	0.00	0.00
445.gobmk	379	61.48	89.18	37.73	46.70	40.37
456.hmmer	245	89.80	81.63	40.41	76.33	60.82
458.sjeng	130	99.23	44.62	45.38	72.31	80.00
462.libquan.	1	0.00	0.00	0.00	100.00	0.00
464.h264ref	89	89.89	73.03	37.08	84.27	66.29
433.milc	43	88.37	97.67	6.98	69.77	65.12
470.lbm	0	0.00	0.00	0.00	0.00	0.00
482.sphinx	16	31.25	31.25	18.75	68.75	12.50
471.omnet.	130	72.31	76.92	22.31	70.77	43.08
473.astar	0	0.00	0.00	0.00	0.00	0.00
483.Xalan	1916	46.03	65.55	21.14	80.64	54.96
444.namd	2	0.00	50.00	50.00	0.00	100.00
447.dealII	1851	18.10	65.80	17.77	64.94	57.16
450.soplex	228	84.65	75.44	26.32	62.72	56.58
453.povray	1602	91.39	38.76	19.48	75.28	16.10
410.bwaves	0	0.00	0.00	0.00	0.00	0.00
416.gamess	3088	79.18	56.99	35.65	73.74	52.91
434.zeusmp	14	0.00	50.00	57.14	71.43	50.00
435.gromacs	360	84.44	87.50	34.44	73.61	35.56
436.cactus.	376	95.48	80.59	56.65	84.31	58.24
437.leslie3d	2	0.00	100.00	100.00	50.00	50.00
454.calculix	269	75.09	87.36	53.53	40.15	37.17
459.Gems.	72	80.56	76.39	50.00	43.06	62.50
465.tonto	1631	90.74	88.90	17.78	41.02	40.34
481.wrf	572	77.45	93.71	27.97	29.55	36.71
Overall	**21360**	**77.44**	**73.32**	**31.75**	**67.02**	**47.17**

Fig. 13. Effects of each checking mechanism

As shown in the figure, the percentage of each function category is largely language and program dependent. For most C and Fortran programs, direct calls contribute to the largest number of identified functions. (Note that this includes direct calls made within functions that are only indirectly reached.) Some C programs (such as 445.gobmk), however, contain a large number of indirect functions. For many C++ programs, because virtual functions[7] are abundant, there are generally more indirectly reached functions. The fourth column presents the percentage of functions that are reached *only* by direct jumps (i.e., tail called). These functions are not rare in optimized binaries.

Note that for some benchmarks, the percentage of unreachable functions is quite high (average 17.7% and up to 40%). To verify these results, we selected a subset of these programs, and used Pintools [27] to record the locations reached via calls or jumps. We found that none of these addresses corresponded to functions determined unreachable by our technique.

After checking source code, we found that the unreachable functions are mostly global (i.e., non-static) functions which are neither called directly nor have their addresses taken. Although they are not used, compilers don't generally remove them unless specific actions are taken during the build process to eliminate them. Note that this is different from static functions whose visibility is within the same compilation unit — it is more common for unused static functions to be removed by default.

[7]Virtual functions can only be *indirectly* called through a V-table.

Effectiveness of Interface Checking Techniques.

As discussed, function interface checking is critical in pruning spurious functions from the identified candidate set. In this section, we evaluate the effectiveness of each checking mechanism independently. The results are presented in Fig. 13. Again, only GCC -O2 compiled binaries for x86-32 are shown.

As shown in the figure, each checking mechanism is independently effective in identifying a significant fraction of all spurious ("total pruned" in the figure) functions. However, in general, no single mechanism is able to detect all spurious functions. It is through their combination that we can effectively reduce the number of spurious functions to a very low number. Note that for four of the binaries, no spurious functions are pruned. This is because all the functions enumerated happen to be real functions.

VII-F. Analysis Runtime Performance

Our focus so far has been on accuracy, and hence we have not made any efforts to optimize runtime performance. Nevertheless, for completeness, we summarize the performance results we currently obtain.

As compared to machine learning based approaches [7, 35], one of the advantages of our approach is that it does not require training, which is expensive. The results of our analysis, together with those from ByteWeight [7] and neural network based system [35], are summarized in Fig. 14. The numbers

Tool	Experiment setup			x86-32 binaries		x86-64 binaries	
	machine	CPU	RAM	training	testing	training	testing
ByteWeight	desktop	4-core 3.5GHz i7-3770K	16G	293 h_c (estimate)	457,997 s	293 h_c (estimate)	593,170 s
Neural	Amazon EC2 c4.2xlarge	8-core 2.9GHz Intel Xeon	15G	20 h	1,062 s	20 h	1,018 s
Ours	laptop	4-core 1.7GHz i5-4210U	8G	0	47,880 s	0	36,300 s

Fig. 14. Experiment setup and performance comparison (h_c = compute hours, h = hours, s = seconds)

are based on our first data set, and 10-fold cross validation for machine learning systems.

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 comparison, 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 40 seconds on average to analyze a binary of our test suite. Although this is already satisfactory 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 and therefore analysis of the whole function can be avoided. This is in contrast to our current naive implementation that performs complete analysis and checks.

VIII. Case Studies

Since function recognition serves as an essential step for many techniques working on binaries, to understand how well our approach can be used, we analyze several representative applications. Our focus in this section is on binary instrumentation tasks that impose stringent requirements.

Many binary instrumentation techniques operate on individual functions as a unit [12, 32, 10]. These applications are sensitive to precision because a misidentified function could cause misbehavior or failure of the instrumented program. In this section, we analyze the applicability of our function identification system for these function-based instrumentation applications.

Since directly called functions are free of errors and unreachable functions are not relevant for correct functionality (as they will never be executed), imprecision could possibly originate from two sources: indirectly reachable functions and direct jump reached (tail called) functions. We analyze these two cases respectively.

For the first case, an address could be incorrectly identified as an (indirectly reachable) function start if our interface checking mechanism is insufficient. Although our comprehensive checking schemes are generally effective and can remove vast majority of the spurious function starts, such misses do happen. Fig. 15 shows one example. In this case, since all instructions access global memory and there are no stack or general-purpose register operations, our interface checking can't identify [818c784, 818c8e4] as a spurious function.

```
0818ba30 <func>:
818ba30:    fld   0x86ed1d0 ;real function start
818ba36:    fstp  0x8ca5af0
...         (similar instructions)
818c784:    fld   0x87202c0 ;spurious function start
818c78a:    fstp  0x8ca5908
818c790:    fld   0x87202c8
818c796:    fstp  0x8ca5910
...         (similar instructions)
818c8d0:    mov   $0xf2,0x8ca5a4c
818c8da:    mov   $0xf6,0x8ca5aec
818c8e4:    ret
```

Fig. 15. A falsely identified (indirectly reachable) function [818c784, 818c8e4]

Despite these imprecisions, one distinguishing feature of our system is that the real function which encloses the spurious one is always identified. In Fig. 15 for example, [818ba30, 818c8e4] is also recovered. And with this property, different measures could be taken for different instrumentations to cope with the imprecisions.

For RAD [12], no work is required at all because the instrumentation is resistant to such imprecisions[8]. For more complicated instrumentations [32, 10], 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 [27, 45, 44, 37]) 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, incorrect instrumentation will not be executed.

Our system may also falsely recognize intra-procedural jumps as direct tail calls (the second type of error). Essentially, this is equivalent to splitting the original function into two. However, we note that this will not introduce any correctness problems, as all executed instructions and exercised control flows have been well captured.

Above analysis indicates that our function recognition is effective and only leaves *limited* error possibility. The inaccuracies tend to either have no effect for function-based instrumentation correctness, or can be easily coped with. As a comparison, since machine learning based approaches rely on code or byte patterns, false positives of function starts and

[8]This is because, at the spurious function start, an extra (i.e., unneeded) "return address" will be pushed on 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 is present in the shadow stack, because the larger, real function is also recovered.

ends are much more random and difficult to deal with. Finally, as can be seen in these case studies, our system automatically classifies identified functions based on their reachability property, which can enable more flexible instrumentations.

IX. Discussion

Special calling conventions. Currently our data flow checking technique is based on well-respected system ABIs and calling conventions, and it can be adapted to other architectures such as ARM [11]. We note although non-standard calling conventions have not appeared in our tests, they could be used in some cases, e.g., function calls within a single translation unit. To deal with this issue, a "self-checking" mechanism can be adopted.

Specifically, note that ABI violations can occur only in the context of direct calls and jumps. (Since a compiler cannot be sure about the target of an indirect control flow transfer, it cannot assume that such a transfer is intra-module.) Since we don't apply interface checks for direct calls, ABI violations won't pose a problem in their context. That leaves direct jumps (i.e., direct tail calls) as the only problem case. We develop a self-checking mechanism in this case. Specifically, we can perform interface checks on a subset of directly called functions to determine whether ABI is respected. If not, we identify a relaxed set of conventions that are respected in direct calls, and apply these relaxed checks to tail call verification. (Note that verification of indirectly called functions can continue to rely on ABI.)

X. Related Work

Function recognition. Many tools recognize functions using call graph traversal and function prologue matching. Examples include CMU BAP [8], angr binary analysis platform [36], and the Dyninst instrumentation tool [22]. IDA [1] uses proprietary heuristics and a signature database for function boundary identification to assist disassembling. Its problems 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 [31]. The precision and performance have been greatly improved by recent work from Bao et al. [7] and Shin et al. [35], due to adoption of different machine learning techniques such as weighed prefix trees and neural networks. However, as discussed, machine learning requires a good training set, and potentially subtle parameter tuning. Moreover, existing machine learning techniques have been focused on surrounding code, and may have difficulties grasping valuable global evidences or deeper semantics — the factors greatly benefit our analysis.

Nucleus [5] is a concurrent work that is also based on static analysis. Nucleus relies on control-flow analysis to infer inter-procedural edges and function starts. In contrast, our approach leverages both control-flow and data-flow properties for comprehensive function interface checking. We demonstrate that *fine-grained* static analysis [29] can recognize functions with much greater accuracy, and has the potential to support demanding applications such as automated analysis and instrumentation.

Static binary analysis to recover high-level constructs. Other than function boundaries, previous works also focus on recovering other high-level constructs, such as variables and types [6, 26, 3] or function signatures [17]. The more ambitious goal is to recover source code through decompilation [21, 18, 33]. However, many of these tools are either best-effort analyses designed for helping human audits, or only tested with a much smaller corpus. We expect the precision of these downstream analyses to be improved with more accurately recognized functions.

XI. 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. By adopting a comprehensive checking mechanism that combines stack discipline, control flow and data flow properties, our approach can substantially improve accuracy over the best previous systems that are either machine learning or static analysis based. The deeper insights of identified functions provide further opportunities to reduce error rates and enable more flexible applications.

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, 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 EuroSys*, 2013.

[4] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security*, 2016.

[5] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *EuroS&P*, 2017.

[6] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you eXecute. *ACM TOPLAS*, 2010.

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

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

[9] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan. Bingo: Cross-architecture cross-os binary search. In *FSE*, 2016.

[10] 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.

[11] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. Azab, L. Lu, H. Vijayakumar, and W. Shen. Norax: Enabling execute-only memory for COTS binaries on AArch64. In *S&P*, 2017.

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

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

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[15] Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. In *PLDI*, 2016.

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

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

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

[19] H. Flake. Structural comparison of executable objects. In *DIMVA*, 2004.

[20] A. Fog. Calling conventions for different c++ compilers and operating systems, 2015.

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

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

[23] N. Hasabnis, R. Qiao, and R. Sekar. Checking correctness of code generator architecture specifications. In *CGO*, 2015.

[24] N. Hasabnis and R. Sekar. Extracting instruction semantics via symbolic execution of code generators. In *FSE*, 2016.

[25] N. Hasabnis and R. Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *ASPLOS*, 2016.

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

[27] 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.

[28] X. Meng and B. P. Miller. Binary code is not easy. In *ISSTA*, 2016.

[29] R. Qiao and R. Sekar. Effective function recovery for COTS binaries using interface verification. Technical report, Secure Systems Lab, Stony Brook University, 2016.

[30] R. Qiao, M. Zhang, and R. Sekar. A principled approach for ROP defense. In *ACSAC*, 2015.

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

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

[33] 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.

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

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

[36] 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.

[37] 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 *WCRE*, 2013.

[38] 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 paper.*, 2008.

[39] 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.

[40] 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.

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

[42] 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.

[43] M. Zhang. *Static Binary Instrumentation with Applications to COTS Software Security*. PhD thesis, Stony Brook University, 2015.

[44] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *VEE*, 2014.

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

[46] M. Zhang and R. Sekar. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In *ACSAC*, 2015.