Towards Optimal Use of Exception Handling Information for Function Detection

Chengbin Pang*†§ Ruotong Yu†§ Dongpeng Xu‡ Eric Koskinen† Georgios Portokalidis† Jun Xu†

*Nanjing University †Stevens Institute of Technology ‡University of New Hampshire

***Abstract*—Function entry detection is critical for security of binary code. Conventional methods heavily rely on patterns, inevitably missing true functions and introducing errors. Recently, call frames have been used in exception-handling for function start detection. However, existing methods have two problems. First, they combine call frames with heuristic-based approaches, which often brings error and uncertain benefits. Second, they trust the fidelity of call frames, without handling the errors that are introduced by call frames.**

In this paper, we first study the coverage and accuracy of existing approaches in detecting function starts using call frames. We found that although recursive disassembly with call frames can maximize coverage, using extra heuristic-based approaches does not improve coverage and actually hurts accuracy. Second, we unveil call-frame errors and develop the first approach to fix them, making their use more reliable.

I. Introduction

Function detection is the process of identifying code regions in binary software that are compiled from source-level functions. Accurate function detection is critical for guaranteeing the correctness and effectiveness of mainstream applications of binary security, ranging from binary code similarity detection [22, 29], legacy-code patching [6, 38–40], shadow stack protection [8], coarse-grained [16, 25, 36, 45] or fine-grained [15, 17, 30, 37, 44] control flow integrity, to code layout randomization [10, 18–21, 28, 41, 42, 44].

The first step of function detection is to identify function entry points, or *function starts* and this is, unfortunately, very challenging. First, symbols in a binary provide the true identity of function starts, but those symbols are normally stripped. Second, binary code is often riddled with complex constructs (*e.g.,* jump tables, tail calls, *etc*) for performance optimization. Mainstream conventional approaches for function start detection [7, 24, 27] first recursively disassemble a given binary from known function starts (*e.g.,* program entry) and add the targets of call instructions as new function starts. They then scan the non-disassembled code to further detect function starts with common function prologues [32, 34] or data-mining models [7, 24], followed by recursive disassembly again. Beyond such a hybrid approach, there are also solutions that either (i) use data-mining models or neural networks to detect function starts [5, 33] or (ii) aggregate basic blocks connected by intra-procedural control flows into groups and consider the target of each call instruction or the first instruction in each group as a function start [4].

§These authors contributed equally to this paper. This work was done while Pang was a Visiting Scholar at Stevens Institute of Technology.

Although the above approaches have demonstrated some effectiveness in detecting function starts, they still share a fundamental drawback: they all attempt to recover function information using a pattern-driven principle, explicitly or implicitly. This drawback impedes the adoption of those approaches in the context of security applications. In fact, the patterns used by them are usually incomplete (missing true function starts) and/or inaccurate (introducing false function starts). Unlike symbols whose reliability is guaranteed by compilers, the patterns collected by these approaches do not build on any reliable source. Inevitably, those approaches lead to errors or omissions, which in turn reduces the confidence of users and even leads to cascading effects.

Recent advances [35, 43] have leveraged a new source to detect function starts in x64 binaries: call frames in the exception handling segment. To support exception handling, compilers emit call frames in x64 binaries as mandated by the ABI, giving information such as the start location for functions wherever possible. Mainstream binary analysis tools, in particular GHIDRA [2] and ANGR [34], already use call frames to facilitate function start detection. However, we observe two common, critical problems. First, the tools try to improve coverage by mixing the use of call frames with additional approaches that are sometimes safe and sometimes unsafe. Safe approaches leverage knowledge from the binary (e.g., symbols), the machine (e.g., instruction set), and/or the ABI (e.g., calling conventions) to provide correctness guarantees. However, unsafe approaches are also involved, which try to use common patterns but typically do not offer assurances of correctness. These unsafe approaches inevitably introduce errors, sabotaging the reliability of call frames and the safe approaches. Moreover, the benefits from unsafe approaches (*e.g.,* whether they can really improve coverage) remain unclear. Second, the tools fully trust the fidelity of call frames. They do not realize that call frames by themselves can also introduce errors and, not surprisingly, do not include any solutions to fix those errors.

In this paper, we inspect the above two problems, aiming to bring new insights towards optimal strategies of using call frames for function start detection.

First, we study the coverage of existing tools, when combining call frames with other methods, and the accuracy of the results produced. To perform the study, we collected 1,395 binaries from both real-world application and the popular benchmarks, and we separately measured the coverage and accuracy of detecting function starts detected by each combination of

approaches. Our key findings are (i) running safe recursive disassembly with call frames can already provide nearly full coverage; (ii) additionally running unsafe approaches from existing tools does not provide meaningful improvement to the coverage but, can introduce plenty of false positives.[1] These bring insights towards both optimal coverage and better reliability in the use of call frames for function start detection.

Second, we systematically unveil and quantify the errors that call frames can introduce. To be specific, we compared the function starts extracted from call frames and the ground truth in our benchmark binaries. We discovered that modern compilers keep separate call frames (*also separate symbols*) for distant parts in a non-contiguous function. When such call frames are directly used for function start detection, they can bring a significant group of false function starts. We also found that existing tools do not provide any solution to handle such false function starts. Following our findings, we develop a new algorithm to fix errors brought by call frames. Our key insight is that distant parts in a non-contiguous function are typically connected via a jump. By checking that the jump between two call frames cannot be a jump between two functions (i.e, the jump cannot be a tail call), we can decide that the two call frames belong to the same non-contiguous function and thus, merge them. Inspired by this insight, we incorporate well-founded, restrictive criteria to detect tail calls, minimizing the chance of reporting false tail calls and ensuring that all missed tail calls are harmless. According to our evaluation, our algorithm can eliminate nearly 95% of the false function starts introduced by call frames, without incurring harmful side effects. Further, all the missed false function starts are due to conservativeness of our implementation choices instead of the design of our algorithm.

Our main contributions are as follows.

- **New knowledge** - We investigate the coverage and accuracy of function starts detected by combing call frames with different approaches from existing tools. We bring insights towards using call frames to achieve optimal coverage of function starts with a minimal hurt to the reliability.
- **New approach** - We are the first to systematically study, classify, and quantify the errors that call frames can bring. We develop the first approach that can fix the errors in call frames, making them a better information source for function start detection.
- **New finding** - We unveil key problems in how existing tools use call frames and demonstrate their significance with quantitative evidence.
- **New tool** - We developed a tool incorporating all our strategies. Its source code is available at https://github.com/ruotongyu/FETCH.

[1]False positive means the start of a function identified but it is actually not. False negative means the start of a function is not identified.

II. Overview

A. Problem Definition

Informally, function detection is to reconstruct the mapping from the code in a binary to the corresponding functions in the source code. At the binary level, a function consists of a set of basic blocks, which has one entry point and one or more exit points. The principled solution of function detection is to find a function entry point firstly, or a *function start*, and then follow the intra-procedural control flow to detect instructions until reaching the exit points. Accurately finding function starts is a universal foundation of function detection. In this paper, *we, therefore, focus on function start detection*.

B. Existing Solutions

Past research has brought many solutions of function start detection. Most solutions developed in the earlier stage use three strategies. First, ByteWeight [5] and Shin *et al.* [33] train decision trees and neural networks to detect function starts from raw binaries. Second, Nucleus [4] first recovers the instructions using linear sweep and then aggregates the instructions connected via intra-procedural control flows into groups. The target of a direct call instruction or the lowest address in each group is considered a function start. Third, the majority of tools (*e.g.,* Dyninst [24], BAP [7], and Radare2 [32]) use a hybrid solution. The tools first gather symbols remaining in the binary and then run recursive disassembly from each symbol. The addresses of the symbols and targets of direct/indirect calls found in recursive disassembly are considered function starts. The tools finally detect function starts in the non-disassembled regions using common prologues or data mining models [27], followed by recursive disassembly from the newly discovered function starts.

A fundamental limitation shared by the above solutions is that, they all heavily rely on pattern matching or empirical learning to recover function starts from binary code. Even with the hybrid solutions, nearly 18% of the function starts are detected by prologue matching (without counting the functions recursively found from those function starts) [27]. The patterns and learned models can be incomplete or inaccurate and oftentimes over-fit the "training" data. As a consequence, those solutions inevitably introduce errors or miss true function starts.

Using Exception Handling Information: Recently, many tools are adopting a more reliable source of information — the exception-handling information — to facilitate function start detection. Both angr [34] and Ghidra [2] leverage call frames in the exception handling section to help detect function starts. They first consider the addresses recorded in existing symbols and call frames as function starts and run recursive disassembly from those addresses to detect more function starts carried by targets of call instructions. Then they take extra steps such as function prologue matching to find further missing function starts. Jima [3] only leverages exception handling information to aid detection of exception handling code blocks.

```
1  double div(int a, int b) {
2     if(b == 0)
3        throw "Division by zero error!";
4     return (a/b);
5  }
6  void main (){
7     int x = (int)getchar();
8     int y = (int)getchar();
9     try {
10       return div(x,y);
11    }
12    catch (const char* msg){
13       cerr << msg << endl;
14    }
15 }
```

Fig. 1: An example of exception handling in C++ programs.

The use of exception handling information by existing tools (ANGR and GHIDRA) in function start detection has two problems. First, they combine the reliable information in call frames with *unsafe approaches*, i.e., approaches that do not offer assurances of correctness: ❶ both ANGR and GHIDRA run prologue matching to detect function starts in the non-disassembled code regions, followed by a round of recursive disassembly from each matched function start; ❷ both ANGR and GHIDRA leverage heuristics to detect tail calls and consider their targets as function starts (*not enabled by default*); ❸ ANGR linearly scans the remaining code gaps and treats the beginning of each correctly disassembled code piece as a new function start [27]. The use of unsafe approaches often bring errors, but it may not increase the coverage achieved by call frames and *safe approaches* that provide correctness guarantees (*e.g.,* recursive disassembly). Second, the existing tools fully trust the fidelity of call frames, without realizing and handling the errors that call frames can bring.

C. Research Scopes

In this paper, we focus on exploring the use of exception handling information for function-start detection. Our goal is not to develop a new approach from scratch. Instead, we aim to expose any shortcomings in how existing tools use exception handling information and identify the best strategies of using exception handling information for function start detection. Specifically, we have the following goals:

- **Goal 1:** We study the coverage of function starts by combining call frames with both safe and unsafe approaches from existing tools. This will bring insights towards optimal coverage with a minimal threat to reliability. § IV discusses how we achieve this goal in detail.
- **Goal 2:** We systematically study the errors that call frames can introduce and explore new solutions to fix the errors. This will help ensure the fidelity of call frames as an information source for function start detection. § V presents our approach to the second goal.

In accordance to our goals, we restrict our discussion in this paper on binaries with call frames. To this regard, we focus on System-V x64 binaries (*e.g.,* x64 binaries running on Linux or other Unix variants) because the corresponding ABI [23] mandates the existence of call frames while the other types of binaries may not have call frames.

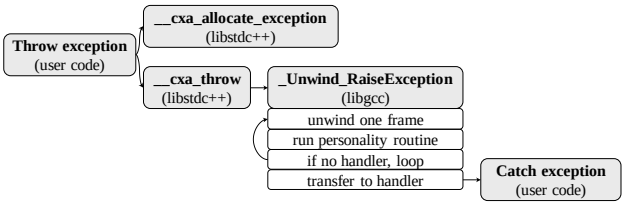


Fig. 2: Workflow of exception handling in C++ programs.

III. Demystifying Exception Handling

In this section, we describe the technical details of exception handling at the binary level and unveil the types of exception handling information that can help function start detection.

A. Exception Handling at the High Level

Exception handling is the process of responding to the occurrence of exceptions during the execution of a program. Support of exception handling has become a standard feature of modern programming languages. For instance, C++ provides the `try`, `throw`, and `catch` clauses to facilitate handling of exceptions. To explain exception handling, we use the C++ example in Figure 1. Exception handling in other programming languages follows a similar format, although using different grammar.

As shown in Figure 1, the `main` function receives two integers from the user and attempts to divide them by calling `div`. In normal cases, `div` returns the division result to `main`, but if the divisor is zero, it throws an exception which will then be caught and handled by `main`. To realize exception handling in this case, execution has to go through two key steps. First, it needs to find the proper handler for the exception. As shown in our example, the throwing of an exception and the suitable handler for that exception can lie in different functions. As such, exception handling may need to search in the call chain on the stack, including the current function where the exception is thrown and all the caller functions. Second, after finding the proper handler, execution is redirected to it.

The above two steps are mainly completed by a special procedure called *stack unwinding*. When an exception occurs, stack unwinding linearly searches every function on the call stack for the exception handler. While searching the exception handler, stack unwinding concurrently updates the stack by removing the stack frame of each searched function until the correct handler is identified. Following that, stack unwinding sets the stack pointer to the frame of the function with the correct handler, recovers the contexts in that function, and switches the execution to that handler.

In Figure 1, once `div` throws the exception at line 3, the execution will in turn search `div` and `main` to locate the right handler at line 12 in `main`. In this process, the execution will remove `div`'s stack frame and then set the stack pointer to `main`'s frame. Finally, the execution will recover the context of `main` and switches to the `catch` clause at line 12.

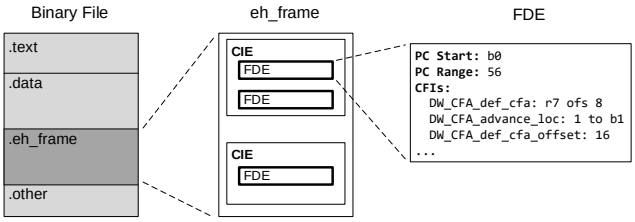


Fig. 3: An overview of the `eh_frame` section.

B. Exception Handling under the Hood

In this section, we further reveal the under-the-hood mechanism of exception handling and stack unwinding. We follow the same setting of our running example: exception handling in x64 binaries compiled from C++ programs.

Figure 2 shows the workflow of the exception handling procedure. We will focus on the part of stack unwinding since other parts are not related to function detection. Stack unwinding is mainly completed by the `_Unwind_RaiseException` function from C library (libgcc). We describe how `_Unwind_RaiseException` performs the stack unwinding procedure as follows. For simplicity, we abbreviate `_Unwind_RaiseException` as F_U.

❶ F_U first checks the program counter (PC), *i.e.,* the `rip` register, at the `throw` statement and determines the current function (*e.g.,* `div` in Figure 1) based on the PC.

❷ F_U then checks if the current function has a proper handler. Specifically, it checks whether the current function has a `catch` block that can handle the `throw`. If a proper `catch` block is found, F_U switches the PC to the `catch` block. Otherwise, F_U recovers the registers saved by the current function and destroys its stack frame by adjusting the stack pointer (SP). Then, F_U goes to the next step.

❸ F_U finds the caller function on the stack (*e.g.,* `main` in Figure 1) and repeats ❷, using the return address as the new PC. However, if the stack frame is empty, F_U will invoke `terminate` to make the program exit abnormally.

As unveiled by the description above, ❶-❸ critically depend on three tasks: ($\mathbb{T}_1$) given PC, finding the function containing the PC; ($\mathbb{T}_2$) given PC and the corresponding SP, determining the call frame of the current function and its return address; ($\mathbb{T}_3$) given PC and the corresponding SP, recovering the registers saved by the current function. To complete the three tasks, F_U leverages information from a special section called `eh_frame`, which is also the key data empowering function detection. In the rest of this section, we will give a brief introduction of `eh_frame` and then explain how it helps complete the three tasks.

C. EH_Frame: Key Data Structure for Exception Handling

Overview of EH_Frame: As illustrated in Figure 3, `eh_frame` is a separate section in a binary file. It is structured as a list of Common Information Entries (CIE), each corresponding to an object file linked into the binary file. A CIE carries one or more Frame Description Entries (FDE), and typically, one FDE records the information of a unique function from the CIE's object file.

Exception Handling with EH_Frame: The major information in `eh_frame` used by exception handling resides in the FDEs. An FDE record consists of a list of fields, among which `PC Begin`, `PC Range`, and `Call Frame Instructions` (CFIs) are indispensable to tasks $\mathbb{T}_1$-$\mathbb{T}_3$. In the following, we will follow the example in Figure 4 to explain how the three fields are used to complete $\mathbb{T}_1$-$\mathbb{T}_3$.

Figure 4a shows the assembly code of a function extracted from IDA-Pro 7.2 and Figure 4b shows the corresponding FDE. Line 2 and 3 in Figure 4b presents the `PC Begin` and the `PC Range` fields in the FDE. They explicitly give the start address and length of the function body. *Using the PC information in the FDEs, exception handling can easily find the function containing a given PC, thus completing task* $\mathbb{T}_1$.

The rest part of Figure 4b (line 4-19) presents `CFIs`, a group of special instructions describing the unwinding rules. Due to historical reasons, the format of `CFI` follows the DWARF standard [9]. The core concept introduced by these unwinding rules is called "Canonical Frame Address (CFA)". CFA is a universal variable that refers to the base address of the current stack frame (typically the highest address), which helps to uniform the various representations of the frame pointer introduced by compilers. There are four main types of instructions involved in the unwinding rules:

- `DW_CFA_def_cfa` defines how CFA is represented, normally in the format of an offset to a designated register.
- `DW_CFA_advance_loc` records the location of an instruction that changes the register used to represent the CFA or saves certain registers to the stack. The instruction location is represented by an offset to the function start.
- `DW_CFA_def_cfa_offset` describes the rule to calculate CFA when the value of the register representing CFA is changed. The rule typically follows the format of an offset relative to that register.
- `DW_CFA_offset` records the saving of certain registers to the stack, covering both the number and the location of the saved register.

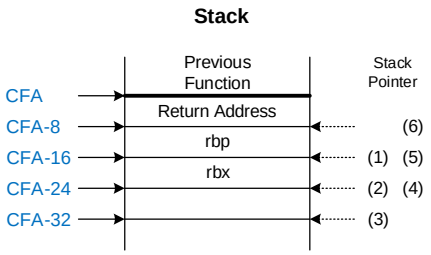
We continue using the example in Figure 4 to explain how the above instructions describe concrete unwinding rules. At line 5 in Figure 4b, a `DW_CFA_def_cfa` instruction defines that `rsp` is used to represent the CFA and initially, `CFA = rsp + 8`. Across the entire function, there are six instructions changing `rsp`, respectively marked as 1-6 in the comments in Figure 4a. Correspondingly, FDE records each change with a separate `DW_CFA_advance_loc` instruction, also marked as 1-6 in the comments in Figure 4b. Following each `DW_CFA_advance_loc` instruction, FDE appends a `DW_CFA_def_cfa_offset` instruction to describe how to re-calculate the CFA. Consider line 2 in Figure 4a as an example. The instruction pushes `rbp` to the stack, decreasing `rsp` by 8. Accordingly, line 6 in Figure 4b indicates that the instruction before address `b1` makes a change to the register representing the CFA (*i.e.,* `rsp`); line 7 in Figure 4b describes that now the offset between CFA and `rsp` is 16 bytes,

```
1  ; function start
2  b0: push rbp ; 1
3  b1: lea rax,[rip+0x36d8b8]
4  b8: lea rbp,[rdi+0x50]
5  bc: push rbx ; 2
6  bd: lea rbx,[rdi+0xb0]
7  c4: sub rsp,0x8 ; 3
8  c8: mov QWORD PTR [rdi],rax
9  cb: nop DWORD PTR [rax]
10 d0: sub rbx,0x18
11 d4: mov rdi, QWORD PTR [rbx]
12 d7: call 3d5c0 <qfree@plt>
13 dc: cmp rbp,rbx
14 df: jne 45fd0 <main+0x55e0>
15 e1: add rsp,0x8 ; 4
16 e5: pop rbx ; 5
17 e6: pop rbp ; 6
18 e7: ret
19 ; function end
```

(a) Assembly code

```
1  00001070 FDE
2   PC Begin: b0
3   PC Range: 56
4   CFIs:
5     DW_CFA_def_cfa: r7 (rsp) ofs 8
6     DW_CFA_advance_loc: 1 to b1 // 1
7     DW_CFA_def_cfa_offset: 16
8     DW_CFA_offset: r6 (rbp) at cfa-16
9     DW_CFA_advance_loc: 12 to bd // 2
10    DW_CFA_def_cfa_offset: 24
11    DW_CFA_offset: r3 (rbx) at cfa-24
12    DW_CFA_advance_loc: 11 to c8 // 3
13    DW_CFA_def_cfa_offset: 32
14    DW_CFA_advance_loc: 29 to e5 // 4
15    DW_CFA_def_cfa_offset: 24
16    DW_CFA_advance_loc: 1 to e6 // 5
17    DW_CFA_def_cfa_offset: 16
18    DW_CFA_advance_loc: 1 to e7 // 6
19    DW_CFA_def_cfa_offset: 8
```

(b) FDE entry from `eh_frame`



(c) The run-time stack

Fig. 4: A function from IDA-Pro 7.2 and its FDE. Addresses of instructions are simplified to only keep the lower two digits.

namely `CFA = rsp + 16`. The rest five `DW_CFA_def_cfa_offset` instructions can be interpreted similarly and the run-time stack is shown in Figure 4c.

The above mechanism guarantees that, given PC and `rsp` at any execution point, CFA can be correctly calculated. This essentially ensures that (i) the range of the current stack frame can be determined since CFA always refers to the highest address of the current stack frame and (ii) the return address of the current function can be found because return address is located right below the top of the current stack frame (*i.e.,* at `CFA - 8`). *Therefore, using information in the CFI, exception handling can correctly complete task* $\mathbb{T}_2$.

Referring back to the example in Figure 4, the first instruction in Figure 4a pushes `rbp`, a callee-saved register, to the stack. Correspondingly, FDE inserts a `DW_CFA_offset` instruction at line 8 in Figure 4b, indicating that the instruction before address `b1` saves register number 6 (*i.e.,* `rbp`) to address `CFA - 16` on the stack. Similar to this case, whenever a callee-saved register is stored on the stack, FDE inserts a `DW_CFA_offset` instruction. As such, *given PC and* `rsp` *at any point of the execution, exception handling can learn what callee-saved registers exist in the current stack frame, and thus, complete* $\mathbb{T}_3$ *in the unwinding process*.

We also note that x64 binaries compiled from C programs similarly carry FDEs, as verified by our studies in § IV. In fact, the x64 ABI mandates FDEs in such binaries as many library functions like `backtrace, __builtin_return_address` need FDEs to support their functionality.

IV. Exploring Coverage of Function Start Detection with Call Frames

In this section, we aim at our first research goal — *exploring the best strategies that use call frames towards optimal coverage of function starts with a minimal harm to the reliability*. We will study the detection of function starts using FDEs with both safe and unsafe approaches from existing tools, and hence, understand which combination of approaches can bring the best balance between coverage and risks. To be more specific, we will center around three questions:

- $\mathbb{Q}_1$ - *Using only FDEs, how many function starts can be detected?*
- $\mathbb{Q}_2$ - *Using FDEs and safe approaches, particularly recursive disassembly, how many function starts can be detected?*
- $\mathbb{Q}_3$ - *Can unsafe approaches, such as the heuristics used by existing tools, help detect more function starts? What are their side effects?*

To answer the above questions, we perform a set of empirical studies with a large corpus of x64 binaries as follows.

A. Setup of Studies

1) Preparation of Datasets: We collected two sets of x64 binaries, one from the wild and one built from source code.

Dataset 1: The first dataset are binaries collected from the wild, including 18 close-source binaries and 25 pre-built binaries from open-source programs. Details of the binaries are shown in Table I. The binaries cover nearly all the common types of software we use in our daily life, ranging from editors to browsers and tele-conference clients. The binaries also cover both C programs and C++ programs.

Dataset 2: The second dataset is compiled from widely-used open source programs. As shown in Table II, the dataset includes 179 programs used by a recent study [27], covering both applications and libraries that are written in C/C++. These programs carry highly diverse functionality and complexity; They also contain both hand-written assembly code and hard-coded machine code. To further increase the diversity, we compiled the 179 programs into x64 binaries with both LLVM (version 6.0.0) and GCC (version 8.1.0), using optimization level O2, O3, Ofast, and Os. We omitted O0 and O1 since they are not widely used in practice. At the end, we produced 1,352 binaries in total.

2) Generation of Ground Truth: To measure the detection results, ground truth about the function starts is required. One common approach to obtaining the ground truth is to use the symbols, but we found that symbols are not perfect: (i) the symbols for hundreds of destructor functions in our dataset are missing; (ii) the symbols for a small group of

TABLE I: Wild binaries in our study. **Open** - open source or not; **EHF** - having `eh_frame` or not; **Sym** - having symbols or not; **FDE** - ratio of functions with FDEs (v.s. symbols).

Software	Open	EHF	Sym	FDE	Note
Atom-1.49.0	✓	✓	✗	–	gcc-7.3.0;c++
Simplenot-1.4.13	✓	✓	✗	–	gcc-4.6.3;c++
OpenShot-2.4.4	✓	✓	✗	–	gcc-4.8.4; c
seamonkey-2.49.5	✓	✓	✗	–	gcc-4.8.5; c++
mupdf-1.16.1	✓	✓	✗	–	gcc-7.4.0; c
laverna-0.7.1	✓	✓	✗	–	gcc-4.6.3; c++
franz-5.4.0	✓	✓	✗	–	gcc-4.6.3; c++
Nightingale-1.12.1	✓	✓	✗	–	gcc-4.7.2; c
palemoon-28.8.0	✓	✓	✗	–	c++
evince-3.34.3	✓	✓	✗	–	c
amarok-2.9.0	✓	✓	✗	–	c
deadbeef-1.8.2	✓	✓	✗	–	c
qBittorrent-4.2.5	✓	✓	✗	–	c++
pdftex-3.14159265	✓	✓	✗	–	c
eclipse-4.11	✓	✓	✗	–	gcc-4.8.5; c
VS Code-1.40.2	✓	✓	✗	–	gcc-7.3.0; c++
VirtualBox-5.2.34	✓	✓	✓	100.0	c++
gv-3.7.4	✓	✓	✓	100.0	c
okular-1.3.3	✓	✓	✓	100.0	c++
gcc-7.5	✓	✓	✓	100.0	c
wkhtmltopdf-0.12.4	✓	✓	✓	100.0	c
firefox-78.0.2	✓	✓	✓	100.0	c++
qemu-system-2.11.1	✓	✓	✓	100.0	c
ThunderBird-68.10.0	✓	✓	✓	100.0	gcc-6.4.0; c++
Smuxi-Server	✓	✓	✓	100.0	gcc-5.3.1; c
TeamViewer-15.0.8397	✗	✓	✗	–	gcc-7.2.0; c++
skype-8.55.0.141	✗	✓	✗	–	gcc-7.3.0; c++
trillian-6.1.0.5	✗	✓	✗	–	c++
opera-65.0.3467.69	✗	✓	✗	–	gcc-7.3.0; c++
yandex-browser-19.12.3	✗	✓	✗	–	gcc-7.3.0; c++
SpiderOakONE-7.5.01	✗	✓	✗	–	gcc-4.1.2; c
slack-4.2.0	✗	✓	✗	–	gcc-7.3.0; c++
rainlendar2-2.15.2	✗	✓	✗	–	gcc-5.4.0; c++
sublime-3211	✗	✓	✗	–	gcc-6.3.0; c++
netease-cloud-music-1.2.1	✗	✓	✗	–	c++
wps-11.1.0.8865	✗	✓	✗	–	c++
wpp-11.1.0.8865	✗	✓	✗	–	c++
wpspdf-11.1.0.8865	✗	✓	✗	–	c++
wpsoffice-11.1.0.8865	✗	✓	✗	–	c++
ida64-7.2	✗	✓	✗	–	gcc-4.8.2; c++
zoom-7.19.2020	✗	✓	✗	–	gcc-4.8.5; c++
binaryninja-1.2	✗	✓	✓	100.0	gcc-5.4.0; c++
FoxitReader-4.4.0911	✗	✓	✓	99.99	gcc-4.8.4; c++
Avg.	-	-	-	**99.99**	-

assembly functions in our dataset have incomplete types. More importantly, symbols can introduce a significant group of false positives, as we will show in § V. Thus, we only use symbols for the pre-compiled binaries in dataset 1 since we have no other options; while for dataset 2, we use a compiler-based approach to produce more complete and more accurate ground truth. Specifics are as follows.

Ground Truth for Dataset 1: As described above, we considered symbols as the ground truth of function starts for the binaries from the wild. Specifically, among the 25 binaries pre-built from open source projects, we found symbols in 1 of them and we successfully installed symbols for 8 others. For the closed-source binaries, we found symbols in 2 of them. In total, we obtained symbol-based ground truth for 11 wild binaries and our studies only considered them.

Ground Truth for Dataset 2: To generate a better ground truth than the symbols, we re-used the frameworks developed by [27] to intercept the end-to-end compiling and linking process to obtain all function starts.

TABLE II: Self-built programs used in our study. **EHF** - having `eh_frame` or not; **FDE** - the ratio of functions that have FDEs, where the baseline is symbols.

Project	Type	# Prog/Bins	EHF	FDE	Lang
Coreutils-8.30	Utilities	105/840	✓	100.0	C
Findutils-4.4	Utilities	3/24	✓	100.0	C
Binutils-2.26	Utilities	17/136	✓	100.0	C/C++
Openssl-1.1.0l	Client	1/4	✓	96.40	C
D8-6.4	Client	1/4	✓	100.0	C++
Busybox-1.31	Client	1/8	✓	100.0	C
Protobuf-c-1	Client	1/6	✓	100.0	C++
ZSH-5.7.1	Client	1/2	✓	100.0	C
Openssh-8.0	Client	7/28	✓	100.0	C
Mysql-5.7.27	Client	1/6	✓	100.0	C++
Git-2.23	Client	1/8	✓	100.0	C
filezilla-3.44.2	Client	1/4	✓	100.0	C++
Lighttpd-1.4.54	Server	1/8	✓	100.0	C
Mysqld-5.7.27	Server	1/6	✓	100.0	C++
Nginx-1.15.0	Server	1/6	✓	98.97	C
Glibc-2.27	Library	1/3	✓	99.97	C
libpcap-1.9.0	Library	1/8	✓	100.0	C
libv8-6.4	Library	1/4	✓	100.0	C++
libtiff-4.0.10	Library	1/8	✓	100.0	C
libxml2-2.9.8	Library	1/8	✓	100.0	C
libprotobuf-c-1	Library	1/8	✓	100.0	C++
SPEC CPU2006	Benchmark	30/223	✓	99.99	C/C++
Total	-	**179/1352**	-	**99.87**	**-**

B. Answering Question $\mathbb{Q}_1$

Comparing with Symbols: In our first study, we extracted the `PC Begin` fields from all FDEs and compared them with symbols. Not surprisingly, FDEs and symbols are highly overlapped. In the 11 wild binaries, FDEs cover 101,882 (99.99%) of the 101,891 symbols. In 9 out of the 11 binaries, FDEs cover all the symbols (see the column of **FDE** in Table I). The results with self-built binaries are similar. In the 1,352 self-built binaries, FDEs cover 1,138,601 (99.87%) of the 1,140,047 symbols. In 1,319 out of these binaries, FDEs cover all the symbols (see the column of **FDE** in Table II).

Comparing with Ground Truth: We further considered the `PC Begin` fields in our self-built binaries as function starts and compared them with the compiler-generated ground truth. In total, the FDEs cover 1,103,832 of the 1,105,278 function starts. Despite the high overall coverage rate (99.87%), *FDEs alone can still leave large coverage gaps in many binaries*. To be specific, FDEs miss function starts in 33 of our self-built binaries and the average number of missed functions is 43.82. In the binary built from Openssl with Ofast, FDEs miss 237 functions.

Through manual analysis, we found that the majority (1,330 out of 1,446) of the functions missed by FDEs are assembly functions. In principle, to comply with the ABI and generate FDE for every assembly function, the developers should write CFI directives [11] manually. However, this only happens in infrastructural projects[2] instead of everywhere, due to the complexity and error-proneness of manually creating unwinding rules. The other functions missed by FDEs are the instances of `__clang_call_terminate`, which are statically linked into the binaries by the Clang compiler.

[2]Example: https://github.com/openssl/openssl/blob/33388b44b67145af2181b1/crypto/aes/asm/aes-x86_64.pl#L608

C. Answering Question $\mathbb{Q}_2$

Following our first study, we then investigate whether recursive disassembly, a widely-used safe approach, can detect the missing function starts. Specifically, we ran the built-in recursive disassembly in both ANGR and GHIDRA, starting from addresses carried by FDEs and symbols. In the course of recursive disassembly, both ANGR and GHIDRA consider targets of call instructions as new function starts. As we focus on the effectiveness of recursive disassembly in this study, we disabled the extra heuristics used by ANGR and GHIDRA for function detection, including the tail call detection used by both tools, the function matching used by both tools, and the linear scan used by ANGR (more details can be found in [27]). To guarantee the accuracy of the experiment result, we only considered the self-built binaries since we have the precise ground truth for them.

Results with GHIDRA: GHIDRA can run all the 1,352 self-built binaries. However, the results are well below expectations. In comparison to solely using FDEs, recursive disassembly by GHIDRA significantly reduced the coverage: the total number of covered function starts dropped from 1,103,832 to 1,088,377; and the number of binaries with non-detected functions increased from 33 to 78. The reduction of coverage is mainly caused by a strategy — *control-flow repairing* — that examines the function start after a non-returning function. If the function start cannot be reached by other control flows, GHIDRA removes that function start. Due to inaccuracy in the detection of non-returning functions and incompleteness in the analysis of control flows [27], control-flow repairing often removes many true function starts, leading to reduced coverage as we observed.

We then conducted a follow-up test where we disabled the control-flow repairing. This time GHIDRA's recursive disassembly demonstrated its effectiveness: in comparison to solely using FDEs, it increased the number of detected functions from 1,103,832 to 1,104,786 and dropped the the number of binaries with non-detected functions from 33 to only 6 (see Figure 5a). While the recursive disassembly by GHIDRA indeed brings more coverage, we found that it is accompanied by another heuristic to detect thunk functions. The heuristic considers a function that starts with a jump to be a thunk function and takes the target of the jump as a new function start. In our test, the heuristic introduced over 400 new false positives and increased the number of binaries that have false positives from 488 to 542.

Results with ANGR: ANGR can only run 1,343 of the self-built binaries because it could not open the remaining 9. Before discussing the results, we want to note an observation that can make a significant difference. Specifically, ANGR marks a special group of functions (or precisely, functions that have a basic block solely consisting of padding instructions) as *alignment*. A recent study [27] suggests excluding the alignment functions for comparison. However, we found that doing so will reduce the coverage but not improve accuracy. Therefore, we preserved all the alignment functions.

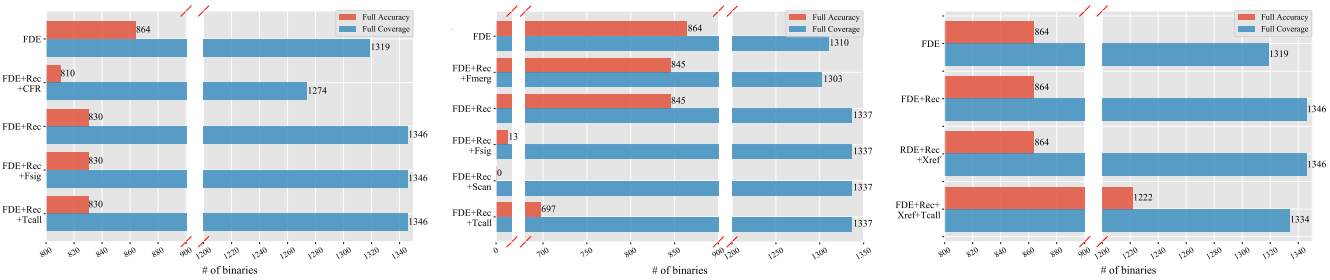
In total, the group of 1,343 binaries contain 982,763 functions. By using FDEs alone, we detected 981,317 of those functions and achieved full coverage in 1310 binaries. However, the further recursive disassembly by ANGR decreased the number of binaries with full coverage to 1,303. The major cause is a heuristic that ANGR uses to merge functions. To be specific, ANGR merges two adjacent functions if the two functions are connected by a jump which is the only outgoing control-transfer from the first function and the only incoming control-transfer to the next function.

Following up the above test, we re-ran ANGR's recursive disassembly without function merging. This time recursive disassembly demonstrated true effectiveness. It increased the number of detected functions from 981,317 to 982,195 and increased the number of binaries with full coverage from 1,310 to 1,337. However, similar to GHIDRA, ANGR's recursive disassembly is coupled with a heuristic that introduces extra false positives. In an alignment function where the beginning instructions are considered padding, the heuristic will mark the first non-padding instruction a new function start, incurring 3,973 false positives.

Results with Safe Recursive Disassembly: As described above, the recursive disassembly by GHIDRA and ANGR is coupled with heuristics. In addition, the recursive disassembly itself in GHIDRA and ANGR also uses other unsafe strategies to handle complex constructs (*e.g.,* indirect jumps) [27]. These indicate that the tests with GHIDRA and ANGR do not truly unveil the coverage of "safe" recursive disassembly on top of FDEs. This motivated us to run an extra test with error-free recursive disassembly. In general, recursive disassembly can run into errors only when handling *indirect jumps*, *indirect calls*, *tail calls*, and *non-returning functions*. We handle these complex constructs as follows to avoid errors.

❶ *Indirect Jumps* - We only consider indirect jumps for jump tables. Specifically, we follow DYNINST [24] to detect and solve jump tables which has proven high precision [27] and fixed some implementation defects in DYNINST.
❷ *Indirect Calls* - We skip all indirect calls.
❸ *Tail Calls* - We do not detect tail calls.
❹ *Non-returning Functions* - We reuse DYNINST's algorithm to detect non-returning functions, which has proven accurate [27]. We expanded the non-returning library functions used by DYNINST to cover all the cases in our self-built binaries. In particular, we handled `error` and `error_at_line` as special cases since they are non-return only when the first argument is non-zero. Encountering either function, we run backward slices from the first argument and examine whether the argument always flows from 0. If so, we consider the function returning and non-returning otherwise.

Running the above error-free recursive disassembly, we achieved identical coverage as GHIDRA and ANGR, while more importantly, we introduced no false positives during the recursive disassembly, as illustrated by Figure 5.



(a) GHIDRA (1,352 binaries) (b) ANGR (1,343 binaries) (c) Optimal Strategies (1,352 binaries)

Fig. 5: The number of binaries where different strategies achieve full coverage or full accuracy. In the figure, **FDE** indicates solely using FDEs; **Rec**, **Fsig**, **Tcall** respectively indicate running recursive disassembly, function matching, and tail call detection in the corresponding tool; **CFR** indicates control flow repairing in GHIDRA; **Fmerg** and **Scan** indicate function merging and linear scan by ANGR; **Xref** indicates function pointer detection in the optimal strategies.

D. Answering Question $\mathbb{Q}_3$

After running our safe recursive disassembly with FDEs, we missed 568 functions in the 1,352 binaries and missed 568 functions in the 1,343 binaries that ANGR can run. All the missed functions are assembly functions, mainly belonging to two groups: (1) functions that are only reachable via tail calls and the successors of those functions (2) functions that are only reachable via indirect calls and the successors of those functions. In the last study, we aim to understand whether the other unsafe approaches used by GHIDRA and ANGR can help detect those functions and what harm they will incur.

Results with GHIDRA: GHIDRA uses two heuristic-based approaches, function matching and tail call detection,[3] to further detect function starts. We tested the two approaches in turn. As indicated by Figure 5a the two approaches are not helpful to coverage. The function matching detected no new function starts, despite it neither brought false positives. The tail call detection found 16 new function starts, however, at the cost of 97,339 new false positives.

Results with ANGR: Besides using function matching and tail call detection, ANGR also detects function starts in its linear-scan process. In our study, we in turn tested function matching, tail call detection, and linear scan. Function matching helped detect 8 new function starts. However, it brought 4,128 false positives and it decreased the number of binaries with full accuracy from 845 to 13; The tail call detection found 211 new function starts at the cost of 4,686 false positives. Moreover, it dropped the number of binaries with full accuracy from 845 to 697; The linear scan detected 230 new function starts. However, it increased the number of false positives from 35,159 to 210,921 and more importantly, it eliminated all the binaries that have full accuracy.

E. Moving Towards Full Coverage

Recursive disassembly with FDEs can provide high coverage, but it does not guarantee full coverage for a specific binary. It would be very convenient if we could identify — or slightly over-approximate — the missing function starts, since this will enable the users to have full coverage and, with slight manual efforts on examining the functions we further identify, obtain a full accuracy (except the accuracy issues inherited from FDEs). To this regard, we explored another soundness-driven approach [31] and we present the details as follows.

Given a binary, we first run recursive disassembly on top of FDEs and then collect all the potential function pointers. For each pointer, we validate its legitimacy. Specifically, we run our conservative recursive disassembly from the pointer and check four types of errors: (i) invalid opcodes; (ii) running into the middle of previously disassembled instructions; (iii) control transfers to the middle of previously detected functions; and (iv) invalid calling conventions (to validate calling conventions, we use the rule that all non-argument registers, namely registers other than rdi, rsi, rdx, rcx, r8, r9, must be initialized before use). If no error occurs, we consider the function pointer legitimate and take it as a new function start.

A key challenge in the above approach is the identification of function pointers. To overcome this challenge, we take a conservative approach to collecting a super-set of function pointers. Technically, we scan every consecutive eight-bytes (*e.g.,* [0,...,7], [1,...,8], [2,...,9], *etc*) in the data segment and the non-disassembled regions, considering each of the eight-bytes as a pointer. We also identify all the constant operands in the disassembled code and consider each constant as a potential pointer. As demonstrated by a recent study [27], this combined strategy can collect all potential function pointers. We further want to note that, once we determine a "legitimate" function pointer, we will update the pointer collection based on the results of recursive disassembly from that pointer.

Applying the above approach to our 1,352 self-built binaries, we detected 154 more function starts without introducing new false positives. We also examined the 414 functions we still missed. These functions belong to two categories. The first category includes 160 unreachable assembly functions (*i.e.,* assembly functions that are not referenced anywhere and their successors). Missing such functions is in principle harmless. The second category contains 254 functions that are only referenced by tail calls in the same function. As we will explain in § V-B, the side effect of missing the 254 functions is equivalent to in-lining them into their parent functions, which

[3]Tail call detection is not enabled by default in GHIDRA (neither in ANGR). We tested it because some missing function starts are due to tail calls.

```
; start of part 1, FDE1
4126c0: push rbp
...
4128ff: test rax,rax
; a non-tail-call jump to part 2
412902: je 404fbe
; end of part 1
; gap
; ...
; start part 2, FDE2
404fbe: mov esi,$0x4437e0
404fc3: xor edi,edi
...
405010: call 4247d0
405015: jmp 404fdc
; end of part 2
```

(a) A non-contiguous function from BinUtils-2.26. The function contains two parts and each part has a separate FDE.

```
; start of function
3c610 <__restore_rt>:
  3c610: mov $0xf,%rax
  3c617: syscall
  3c619:

; FDE Entry
00021e98 FDE
  PC Begin: 3c60f
  PC Range: a
  CFIs:
    DW_CFA_expression: reg8 DW_OP_breg7 +40
    DW_CFA_expression: reg9 DW_OP_breg7 +48
    DW_CFA_expression: reg10 DW_OP_breg7 +56
    ...
    DW_CFA_nop:
```

(b) A handwritten function in Glibc-2.27. The begin of handwritten FDE does not equal to the begin of function start.

Fig. 6: Examples of false positive introduced by FDEs.

is also in general harmless. We finally want to note that while our function pointer detection is not theoretically safe, on average it only reports 0.31 function starts for each binary, which requires minor human efforts to validate.

V. Improving Accuracy of Function Start Detection with Call Frames

In this section, we aim at our second research goal — *understanding the accuracy of function start detection with exception handling information*. Existing tools, GHIDRA and ANGR, simply trust the fidelity of FDEs when using them for function detection. However, we found that FDEs are not perfectly accurate and in fact, they can introduce many false positives. In the following, we will first unveil and quantify the false positives that FDEs can introduce, and then we will present a new approach to fix the errors.

A. Identifying and Quantifying Errors Due to FDEs

To systematically understand the errors introduced by FDEs, we compared the function starts extracted from FDEs in our self-built binaries with the ground truth. We found that FDEs brought 34,772 false positives, spanning 488 of the 1,352 binaries. In the case of Mysqld compiled with GCC and Ofast, FDEs introduced 3, 616 false positives.

Among all the false positive, 34,769 are related to non-contiguous functions. For each non-contiguous function, the compiler inserts separate FDEs for different parts in the function (*e.g.,* Figure 6a). As such, function starts extracted from FDEs for the non-beginning parts become false positives. Such false positives are rooted from the design of FDEs: a single FDE cannot cover multiple non-contiguous code segments. Without adapting the design and the standard behind, it is unlikely to fully avoid such false positives. *We also want to note that symbols have the same problem: separate symbols are generated for different parts from the same non-contiguous function. Our study shows that symbols also introduce the 34,769 false positives.* The remaining 3 FDE false positives are from assembly functions where the developers (intentionally) insert CFI directives that label the incorrect function starts (*e.g.,* Figure 6b).

Algorithm 1 Tail-call Detection and Function Merging

Input: A list of functions L
function TAILCALL-DETECT(L)
  for $f \in L$ **do**
    for direct/conditional jump $j \in f$ **do**
      $t \leftarrow$ Target(j)
      isTailCall = False
      if $t \in f$ **then**
        continue ▷ Skip jump inside function
      end if
      $h \leftarrow$ getStackHeight(j)
      if $h = 0$ **then**
        if HasRefTo(t, L) $\wedge$ MeetCallConv(t) **then**
          add_tail(t) ▷ Find a tail call
          add_func(L, t)
          isTailCall = True
        end if
      end if
      if $\neg$isTailCall $\wedge$ IsFunction(t) $\wedge$ RefTo(t) == j **then**
        MergeFunc(t, f) ▷ Merge function
        remove_func(L, t)
      end if
    end for
  end for
  return L
end function

The false positives brought by FDEs can hurt security applications. For instance, many control flow integrity solutions for binary code [16, 25, 36, 45] consider all function starts as legitimate targets of indirect control transfers. Our experiment with ROPgadget [1] shows that the basic blocks at the FDE-introduced false function starts contain 99,932 valid ROP gadgets. Including the FDE-introduced false function starts would make control hijacks to those ROP gadgets undetectable, reducing the security effectiveness of control flow integrity. However, we observe that both GHIDRA and ANGR do not provide any solution to address the false positives introduced by FDEs: all the the false positives due to FDE persist across every detection step of GHIDRA and ANGR.

B. Fixing Errors Due to FDEs

To effectively reduce the FDE-introduced false positives, we propose a new algorithm based on a key observation that *two distant parts in the same non-contiguous function are connected via a jump*. In principle, by determining that the jump is not a jump between two functions (*i.e.,* not a tail call), we can safely merge the two distant parts. In general, it is hard to design a perfect algorithm to detect tail calls. Most existing tools use heuristics that are neither sound nor complete [27]. In this work, we propose a completeness-driven, but safe, algorithm, which ensures fidelity of the captured cases and minimizes the side effects of the missed cases.

Our algorithm is shown in Algorithm 1. It iterates over each conditional or unconditional jump in each function. It considers a jump to be a tail call if:

❶ The stack pointer at the jump site is right below the return address. This is a must-be-true property of tail call because the current function has to ensure that the target can directly return to its parent function.

❷ The target satisfies the calling conventions since the target of a tail call must be a new function. To validate calling conventions, we re-use the rule as described in § IV-E.

❸ The target is not referenced elsewhere other than jumps in the current function. In theory, a tail call does not have to meet this rule. However, our empirical studies with a large-corpus binaries (listed in Table II) show that this rule can perfectly avoid false positives. More importantly, this rule ensures that the target of any missed tail call is not referenced elsewhere. Thus, the side effect of the missed tail call is equivalent to in-lining the target function to the source function, which should be generally harmless.

For a jump that we determine to be not a tail call, we check whether the target has an FDE record and whether the target is not referenced elsewhere. If both conditions hold, we consider the jump part and the target part are from the same function and we merge the two parts to the same function.

To implement Algorithm 1, there are two challenges. First, it needs to know the value of the stack pointer at a jump site. Many tools, such as DYNINST [24] and ANGR [34], include static analysis of stack height. However, as shown in Table IV, these analyses can often provide inaccurate stack height due to side effects of other errors and defects of engineering. To address this challenge, we opt to use the stack height recorded by CFIs in FDEs. For conservativeness, we only pick functions whose CFIs give complete information of stack height, by checking (i) whether the CFA in the CFIs is represented by `rsp` and the CFA is initialized as `rsp+8` and (ii) whether a `DW_CFA_def_cfa_offset` instruction exists wherever the stack height is changed. We skip functions with incomplete stack height. Second, the algorithm needs to collect all the references to functions. We overcome this challenge by using the conservative approach in § IV-E.

We also looked at the 3 false positives introduced by the developers. We found that the code blocks pointed to by those FDEs all present invalid calling conventions (using the rules in § IV-C). By checking the calling conventions of each function directly identified from FDEs, we detected the three false positives. After removing the false positives and re-running our pointer-based detection in § IV-E, we also identified the false negatives masked by those three false positives.

C. Algorithm Evaluation

We tested the performance of Algorithm 1 with our 1,352 self-built binaries. On top of FDEs, we first ran our recursive disassembly and our function pointer detection. Then, we ran Algorithm 1 and measured the change of both coverage and accuracy. In total, our algorithm reduced the number of FDE-introduced false positives from 34,772 to 2,659, increasing the number of binaries with full accuracy from 864 to 1,222.

Among the remaining 2,659 false positives, 2,656 are still caused by non-contiguous functions. Our algorithm missed detecting them because CFIs in those functions do not provide complete stack height information, and thus, we skipped processing those functions. While intuition suggests we can re-use static analyses from existing tools (*e.g.,* ANGR and DYNINST) for stack height information in such functions, we opted not to do so. The reason, as aforementioned, is that the static analyses can be incomplete or inaccurate. To validate our choice, we also conducted an empirical evaluation. Specifically, we compared the stack height information from CFIs and the stack height information provided by both ANGR and DYNINST. It is worth noting that we only ran the comparison on functions whose CFIs provide complete stack height information. As shown by the results in Table IV, the stack height analyses by ANGR and DYNINST carries both incompleteness and inaccuracy (even just considering the jump sites), using of which can hurt our tail call detection.

We finally examined whether Algorithm 1 brought false negatives and false positives. It turns out that the algorithm did not bring extra false positives, but it introduced 161 new false negatives, slightly reducing the number of binaries with full coverage from 1,346 to 1,334 (see Figure 5c). All the 161 false negatives are because we merged targets of true tail calls to the call sites. Despite missing the 161 functions slightly affects coverage, the 161 functions are only referenced by tail calls in a single function (otherwise they will be detected by our algorithm). In this sense, the side effect of missing those functions is equivalent to in-lining them to their parent functions, which in general produces no harm. This is also the reason why the 254 false negatives we discussed in § IV-E are harmless.

VI. COMPARING WITH OTHER APPROACHES

We finally conducted an extra comparison, where we compared the optimal strategies of using FDEs and 8 existing tools (using the 1,352 self-built binaries shown in Table II). For simplicity of presentation, we will use **FETCH** (**F**unction d**ET**ection with ex**C**eption **H**andling) to represent our optimal strategies of using FDEs.

Setup: FETCH works by first extracting FDEs and then running our safe recursive disassembly (§ IV-B), our function pointer detection (§ IV-E), and our tail call detection (§ V-B). The 6 other tools that do not use FDEs are from two categories: (i) open-source tools that are designated for function detection or have a component of function detection (DYNINST [24], BAP [7], RADARE2 [32], and NUCLEUS [4], and (ii) commercial tools that can detect functions (IDA PRO [14] and BINARY NINJA [26]). Their configures are same as [27]. The results of GHIDRA and ANGR are also included for convenience of comparison.

Coverage and Accuracy: As shown in Table III, FETCH presents extremely high coverage and accuracy. It only brings hundreds (or dozens) false positives and false negatives from the total 1,352 binaries, regardless of the optimization level. FETCH outperforms all the 8 other tools. It produces the best coverage in all the settings and the best accuracy except under optimization level Ofast. These results demonstrate the benefits of the FDE-assisted solutions.

Efficiency: We also measured the average time required by each tool to run a binary, and we show the results in Table V. Overall, FETCH can finish analyzing a binary in around 3.3 seconds, which represents a high efficiency.

TABLE III: Comparison results between FETCH and existing tools. The results highlighted in blue indicate the best results. FP #: the number of false positives (*thousands*); FN #: the number of false negatives (*thousands*).

OPT	DYNINST		BAP		RADARE2		NUCLEUS		IDA PRO		BINARY NINJA		GHIDRA		ANGR		FETCH	
	FP #	FN #	FP #	FN #	FP #	FN #	FP #	FN #	FP #	FN #	FP #	FN #	FP #	FN #	FP #	FN #	FP #	FN #
O2	12.20	81.41	148.94	93.82	4.10	100.23	17.49	18.41	2.68	37.04	41.17	11.96	43.49	5.62	51.68	0.20	0.78	0.08
O3	12.60	82.05	165.06	96.71	4.26	106.99	20.15	16.70	2.71	36.94	44.57	12.80	46.62	4.65	54.71	0.19	0.84	0.08
Os	6.72	87.82	109.50	79.86	3.08	81.67	23.35	27.86	0.97	32.74	34.45	6.14	0.92	1.97	37.10	0.18	0.07	0.14
Of	13.63	88.22	159.41	92.20	3.08	93.93	26.68	19.36	0.86	37.97	40.08	10.39	46.43	4.66	67.43	0.18	0.97	0.12
Avg.	**11.29**	**84.88**	**132.48**	**90.65**	**3.63**	**95.71**	**21.92**	**20.58**	**1.81**	**36.17**	**40.07**	**10.32**	**34.37**	**5.23**	**52.73**	**0.19**	**0.67**	**0.11**

TABLE IV: Coverage and precision of stack height analyses by ANGR and DYNINST. Baseline is the stack height from CFIs. **Full** indicates the result with all code locations and **Jump** indicates the result with only jump sites considered.

OPT	ANGR				DYNINST			
	Full		Jump		Full		Jump	
	Pre	Rec	Pre	Rec	Pre	Rec	Pre	Rec
O2	93.00	97.26	98.56	95.95	93.18	98.26	98.60	99.36
O3	93.66	97.28	98.62	95.84	92.87	97.96	98.50	99.34
Os	96.42	99.22	99.27	99.27	99.10	98.27	98.67	99.35
Of	93.20	97.08	98.43	94.53	94.08	98.60	98.90	99.36
Avg.	**94.07**	**97.71**	**98.72**	**96.40**	**94.81**	**98.27**	**98.67**	**99.35**

TABLE V: Average time needed by tools to run a binary.

Tool	DYNINST	BAP	RADARE2	NUCLEUS	GHIDRA	ANGR	IDA	NINJA	FETCH
Avg.	2.8s	114.2s	34.9s	3.1s	40.4s	78.5s	10.3s	20.4s	3.3s

VII. DISCUSSION

In this section, we discuss the limitations and future directions of our research.

A. Threats to Validity

In this research, we focus on exploring the best strategies to (i) achieve optimal coverage and accuracy of using FDEs for function start detection and (ii) minimize the risks to the reliability. There exists potential threats to the fidelity of our findings. First, we concluded that running safe recursive disassembly on top of FDEs can achieve extremely high coverage with guaranteed reliability. However, recursive disassembly in practice may not ensure safety due to complex constructs like indirect jumps and non-returning functions. To reduce this threat, we have adopted the most conservative strategies to handle them (§ IV-C). Second, the reliability of our approach to fixing FDEs-introduced errors is threatened by the completeness of the algorithm of tail call detection. To mitigate this threat, we adopted three restrictive criteria to detect tail calls, which have empirically proven completeness (§ V-B). Finally, as we unveiled in § V-A, developers may manually insert or modify the contents of `eh_frame` (intentionally or unintentionally) in a way that introduces errors. This is a threat to the accuracy and we currently cannot avoid the threat. However, such errors rarely happen in practice and therefore, we envision it would not raise major concerns.

B. Generality of Study

Our study focuses on x64 System-V binaries because such binaries are guaranteed by the ABI to have call frames. However, this does not mean our study cannot be applied to other types of binaries. In fact, the methods used in our study are architecture independent and can work with any types of binaries that have call-frame-similar data structures. We have already conducted preliminary studies on other types of binaries and confirmed the availability of a structure similar to call frames. In particular, we found that x86 System-V binaries also widely carry FDEs, covering nearly all the functions . We also discovered that x64 PE binaries adopt an FDE-similar data structure to support exception handling [13], which contains the starts and boundaries of functions. Our preliminary results show that at least 70% of the functions are covered by this structure. In addition, the ABI of Arm architecture also has the similar structure to support exception handling [12]. As a future work, we plan to extend our study to cover other types of binaries.

VIII. CONCLUSION

In this paper, we focus on studying the use of call frames to detect function starts. We found that the use of call frames by existing tools has two common problems. First, beyond using call frames and safe approaches, existing tools also run additional unsafe approaches to detect function starts, seeking to improve the coverage. However, the unsafe approaches can often introduce errors and their capacity of improving coverage is unclear. Second, the existing tools fully trust the information from call frames, without recognizing that call frames can also introduce errors. To gain a deeper understanding of the two problems and hence, bring insights towards optimal strategies of using call frames for function start detection, we conducted two studies. In the first study, we measured the coverage and accuracy of function starts detected by different approaches that existing tools run on top of call frames. Our key finding is that combing safe recursive disassembly and call frames can already achieve the maximal coverage, and additionally including other unsafe approaches cannot benefit the coverage but can hurt the accuracy and reliability of the results. In the second study, we systematically unveiled and quantified the errors that call frames can introduce. We further presented the first approach that can effectively fix nearly all the errors without introducing side effects.

ACKNOWLEDGMENTS

We would like to thank our shepherd Miklos Telek and the anonymous reviewers for their feedback. This project was supported by the Office of Naval Research (Grant#: N00014-16-1-2261, N00014-17-1-2788, and N00014-17-1-2787) and NSF (Grant#: CNS-1948489). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

REFERENCES

[1] "Ropgadget," http://shell-storm.org/project/ROPgadget/, 2011.

[2] N. S. Agency, "Ghidra," https://www.nsa.gov/resources/everyone/ghidra/, 2019.

[3] J. Alves-Foss and J. Song, "Function boundary detection in stripped binaries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 84–96.

[4] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 177–189.

[5] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "{BYTEWEIGHT}: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium*, 2014, pp. 845–860.

[6] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. ACM, 2011, pp. 9–16.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

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

[9] D. D. I. F. Committee *et al.*, "Dwarf debugging information format, version 4," 2010.

[10] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming." in *NDSS*, 2015.

[11] G. Developers, "Cfi directives," https://sourceware.org/binutils/docs/as/CFI-directives.html#CFI-directives, 2020.

[12] A. Docs, "Exception handling abi for the arm architecture - abi 2018q4 documentation," https://developer.arm.com/documentation/ihi0038/latest/, 3 2021.

[13] M. Docs, "x64 exception handling," https://docs.microsoft.com/en-us/cpp/build/exception-handling-x64?view=vs-2019.

[14] C. Eagle, *The IDA pro book*. No Starch Press, 2011.

[15] M. Elsabagh, D. Fleck, and A. Stavrou, "Strict virtual call integrity checking for c++ binaries," in *2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 140–154.

[16] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *7th USENIX Security Symposium*, 2006, pp. 75–88.

[17] W. He, S. Das, W. Zhang, and Y. Liu, "No-jump-into-basic-block: Enforce basic block cfi on the fly for real-world binaries," in *54th Annual Design Automation Conference 2017*. ACM, 2017, p. 23.

[18] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *2012 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 571–585.

[19] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 461–477.

[20] H. Koo and M. Polychronakis, "Juggling the gadgets: Binary-level code randomization using instruction displacement," in *11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 23–34.

[21] L. Li, J. E. Just, and R. Sekar, "Address-space randomization for windows systems," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 2006, pp. 329–338.

[22] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 667–678.

[23] H. Lu, M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System v application binary interface," *AMD64 Architecture Processor Supplement*, 2018.

[24] X. Meng and B. P. Miller, "Binary code is not easy," in *25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 24–35.

[25] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert, "τcfi: Type-assisted control flow integrity for x86-64 binaries," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 423–444.

[26] B. Ninja, "binary.ninja," https://binary.ninja/, 2019.

[27] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *42nd IEEE Symposium on Security and Privacy (SP)*, 2021.

[28] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 601–615.

[29] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 709–724.

[30] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries." in *NDSS*, 2015.

[31] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in cots binaries," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 201–212.

[32] radare, "radare2: Unix-like reverse engineering framework and commandline tools," https://github.com/radare/radare2, accessed Aug 9, 2019.

[33] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium*, 2015, pp. 611–626.

[34] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[35] I. Skochinsky, "Compiler internals: Exceptions and rtti," *Recon*, 2012.

[36] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, 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 *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[37] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary code continent: Finer-grained control flow integrity for stripped binaries," in *31st Annual Computer Security Applications Conference (ACSAC'15)*. ACM, 2015, pp. 331–340.

[38] P. Wang, Shuai Wang and D. Wu, "Uroboros: Instrumenting stripped binaries with static reassembling," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 236–247.

[39] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again." in *NDSS*, 2017.

[40] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *24th USENIX Security Symposium*, 2015, pp. 627–642.

[41] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[42] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 367–382.

[43] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 133–147.

[44] 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 *2013 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2013, pp. 559–573.

[45] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *22nd USENIX Security Symposium*, 2013, pp. 337–352.