Artifacts Available V1.1
Artifacts Evaluated Functional V1.1

OCFI: Make Function Entry Identification Hard Again

Chengbin Pang*
State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology
Nanjing University, Nanjing, China

Tiantai Zhang*
State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology
Nanjing University, Nanjing, China

Xuelan Xu
State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology
Nanjing University, Nanjing, China

Linzhang Wang
State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology
Nanjing University, Nanjing, China

Bing Mao
State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology
Nanjing University, Nanjing, China

ABSTRACT

Function entry identification is a crucial yet challenging task for binary disassemblers that has been the focus of research in the past decades. However, recent researches show that call frame information (CFI) provides accurate and almost complete function entries. With the aid of CFI, disassemblers have significant improvements in function entry detection. CFI is specifically designed for efficient stack unwinding, and every function has corresponding CFI in x64 and aarch64 architectures. Nevertheless, not every function and instruction unwinds the stack at runtime, and this observation has led to the development of techniques such as obfuscation to complicate function detection by disassemblers.

We propose a prototype of OCFI to obfuscate CFI based on this observation. The goal of OCFI is to obstruct function detection of popular disassemblers that use CFI as a way to detect function entries. We evaluated OCFI on a large-scale dataset that includes real-world applications and automated generation programs, and found that the obfuscated CFI was able to correctly unwind the stack and make the detection of function entries of popular disassemblers more difficult. Furthermore, on average, OCFI incurs a size overhead of only 4% and nearly zero runtime overhead.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; *Software security engineering*.

KEYWORDS

function entry detection, obfuscation, binary disassembly

ACM Reference Format:
Chengbin Pang, Tiantai Zhang, Xuelan Xu, Linzhang Wang, and Bing Mao. 2023. OCFI: Make Function Entry Identification Hard Again. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598097

1 INTRODUCTION

Stack unwinding is a crucial process used in debugger and exception handling that involves removing function entries from the function call stack at runtime. To achieve this, the System V ABI in x86 architecture reserves a register `ebp` to represent the base of stack frame [25]. Although it is convenient to walk the stack with the help of `ebp`, it incurs significant overheads due to several reasons. Firstly, one general register is reserved at all times, which is rare in x86 architecture which only has 8 general registers. Secondly, every function requires a specific prologue and epilogue to save and restore the base of the previous stack frame. To overcome the above problems, the `.eh_frame` section has been defined, which encodes the frame pointer and saved registers of every function with the *DWARF* format in x86_64 architecture [27]. Section `.eh_frame` contains call frame information (CFI) which represents a table for every address in binary code, defining how to set registers to restore the previous stack and reveal the range of every function. For more details on the background, please refers to section §2.

Identifying function entries plays a critical role in reverse engineering [3, 6, 33, 34]. Binary disassemblers have found the secret of CFI and leveraged it as the "oracle" of function entry [1, 2, 34]. Recent works show that disassemblers could recover nearly 100% of functions correctly with the help of `.eh_frame` section [33, 34]. Moreover, we studied popular disassemblers (Ghidra [1] and angr [2] which leverage `.eh_frame` to detect function entries) to disassemble the binaries with and without `.eh_frame` section on the x64 dataset presented in [33, 35]. The result is shown in Figure 1. From the result, we find that `.eh_frame` section could improve the accuracy (F1-score [41]: $\frac{2*Precision*Recall}{Precision+Recall}$) of function entries identification significantly on high optimization levels. Furthermore, the use of `.eh_frame` may have implications for malicious purposes, such as software plagiarism [26], malware camouflage [39, 47], and vulnerability exploitation [8, 14, 17, 44].

Binary obfuscation is widely studied to obstruct reverse engineering, such as encoding [42, 54], bogus insertion [10], opaque predicates [10, 51, 55], and control flow flattening [9, 19]. However,

---

*These authors contributed equally.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07...$15.00
https://doi.org/10.1145/3597926.3598097

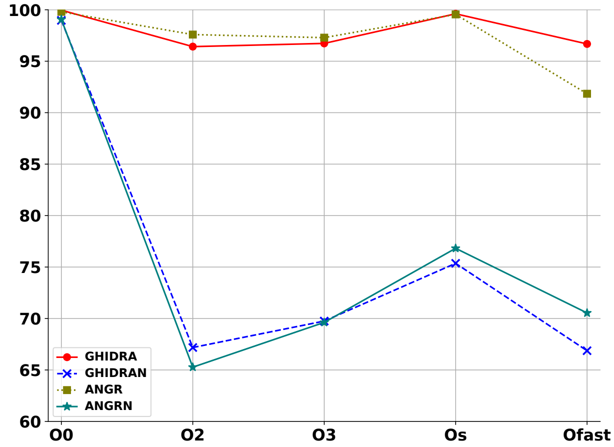


Figure 1: The F1 scores of function entries detection with and without `.eh_frame` section among different optimizations. GHIDRA and ANGR indicate the result with `.eh_frame` section. GHIDRAN and ANGRN indicate the result without `.eh_frame` section.

transformation or encryption in the above works brings extra overhead compared with the original binary code and does not work on CFI directly.

We discovered that not every address in binary code requires stack unwinding, providing insight into obfuscating/debloating the CFI. In this paper, we propose a prototype of obfuscating CFI to obstruct function detection of popular disassemblers that leverage `.eh_frame` section. This approach entails removing unnecessary CFI while retaining the necessary information to ensure correct stack unwinding. However, this approach presents three challenges for obfuscation.

- *C1 – How to determine if a function may unwind the stack at runtime?*
- *C2 – How to minimize the range represented by CFI if this function may unwind the stack?*
- *C3 – How to debloat the CFI to unwind the stack correctly according to the minimized range?*

To address *C1*, we observe that popular compilers, such as GCC and Clang/LLVM, mark known functions that do not unwind the stack as `nounwind`. Based on the observation, we perform backward propagation on the call graph from the known `nounwind` functions to other functions. This approach enabled us to mark other functions that do not unwind the stack as `nounwind`.

To address *C2*, we iterate over every function instruction to determine if it could unwind the stack. Specifically, we check if the instruction is a direct call and if the called function is marked as nounwind. If not, we concluded that the instruction could unwind the stack. As the targets of indirect calls are not easily determined statically, we conservatively assume the indirect call may unwind the stack. Afterward, we mark the first and last instructions that could unwind the stack as the range of CFI.

To address *C3*, we need to determine the correct registers represented in the minimized CFI. Specifically, we perform dataflow analysis to calculate the initialized saved registers and virtual frame pointer inside the minimized CFI.

We implemented OCFI[1] using Clang/LLVM 12.0 to obfuscate CFI. To evaluate the availability and effectiveness of OCFI, we built a large-scale dataset that covers benchmarks, C/C++ real-world applications, and automated generating programs. Based on the evaluation, we found that ① the obfuscated binaries could unwind the stack correctly and ② the obfuscated binaries could obstruct function detection of popular disassemblers (resulting in an average decrease of 37.3% in F1 Score).

The main contributions of this paper are summarized as follows:

- We present the idea of obfuscating CFI without harming the functionality of unwinding stack.
- We present USMITH which generates C/C++ programs with try/catch automatically and could be leveraged to test the availability of exception handling.
- We implement the prototype for OCFI and USMITH and publish the source code at https://github.com/NJUSeclab/OCFI for future research.
- We evaluate OCFI thoroughly and demonstrate the availability and effectiveness of OCFI.
- We test the cost of obfuscation for OCFI, which shows only 4% size overhead and nearly zero runtime overhead on average.

In the rest of this paper, we focus on the application of our obfuscating CFI idea to build OCFI. In §2, we give the technical background of CFI and discuss the necessary CFI for every function to unwind the stack correctly. In §3, we discuss the research scope of OCFI. In §4, we explain the design of OCFI. In §5, we detail the implementation of OCFI. In §6, we present the evaluation of OCFI. §7 discusses our limitations and future works. §8 summarizes the related works and we conclude this paper in §9.

2 TECHNICAL BACKGROUND

In this section, we provide an introduction to the basics of unwinding the stack with call frame information (CFI). This foundational knowledge is essential for understanding how to obfuscate CFI. We also discuss the background of function entry identification by popular disassemblers, providing context for the need to obfuscate this information.

2.1 Exception Handling

Section `.eh_frame` is designed for stack unwinding. When a function is called, the CFI is stored in a data structure known as the Exception Handling Frame (EH Frame), which is located in the .eh_frame section. When an exception is thrown, the function `_Unwind_RaiseException` in libgcc is called to process the exception. We will detail the process as follows.

- *Step-1:* When an exception is thrown, libgcc first checks the program counter (PC) at the throw site, and iterates every FDE (Frame Description Entry) in `.eh_frame` section to find the current FDE based on the PC. FDE is the basic unit of `.eh_frame`. Normally, every function has the corresponding FDE, which consists of the range of the function, CFI, and augmentations (if any). The augmentations allow a language-specific data area

---

[1] OCFI is the abbreviations of Obfuscating Call Frame Information.

```
1  406200: push  %r15
2  406202: push  %r14
3  406204: push  %rbx
4  406205: test  %rsi,%rsi
5  406208: je    40625a
6  ...
7  406239: callq 406270
8  40623e: mov   %r14,%rdi
9  406241: callq 403600
          <_ZdlPv@plt>
10 ...
11 406258: jmp   406210
12 40625a: pop   %rbx
13 40625b: pop   %r14
14 40625d: pop   %r15
15 40625f: retq
16 406260: mov   %rax,%rdi
17 406263: callq 4060b0
          <__clang_call_terminate>
```

(a) Assembly code

```
1  FDE pc=406200..406268
2    DW_CFA_def_cfa: r7 (rsp) ofs 16
3    DW_CFA_advance_loc: 2 to 406202
4    DW_CFA_offset: r16 (rip) at cfa-8
5    DW_CFA_advance_loc: 2 to 406204
6    DW_CFA_def_cfa_offset: 24
7    DW_CFA_advance_loc: 1 to 406205
8    DW_CFA_def_cfa_offset: 32
9    DW_CFA_offset: r3 (rbx) at cfa-32
10   DW_CFA_offset: r14 (r14) at cfa-24
11   DW_CFA_offset: r15 (r15) at cfa-16
12   DW_CFA_advance_loc: 86 to 40625b
13   DW_CFA_def_cfa_offset: 24
14   DW_CFA_advance_loc: 2 to 40625d
15   DW_CFA_def_cfa_offset: 16
16   DW_CFA_advance_loc: 2 to 40625f
17   DW_CFA_def_cfa_offset: 8
18   DW_CFA_advance_loc: 1 to 406260
19   DW_CFA_def_cfa_offset: 32
```

(b) FDE entry from EH_FRAME

PC	CFA	rbx	r14	r15	rip
406200	rsp+16	–	–	–	
406202	rsp+16	–	–	–	*(cfa-8)
406204	rsp+24	–	–	–	*(cfa-8)
406205	rsp+32	*(cfa-32)	*(cfa-24)	*(cfa-16)	*(cfa-8)
40625b	rsp+24	*(cfa-32)	*(cfa-24)	*(cfa-16)	*(cfa-8)
40625d	rsp+16	*(cfa-32)	*(cfa-24)	*(cfa-16)	*(cfa-8)
40625f	rsp+8	*(cfa-32)	*(cfa-24)	*(cfa-16)	*(cfa-8)
406260	rsp+32	*(cfa-32)	*(cfa-24)	*(cfa-16)	*(cfa-8)

(c) Unwinding table

Figure 2: A function from Gold-2.30 and its FDE. (c) is the unwinding table according to the FDE, `rip` represents the return address of the previous caller function.

(LSDA) and `personality` routine to be associated with every FDE.

- *Step-2:* If current FDE contains LSDA and `personality` routine, the libgcc would call the `personality` routine to interpret the LSDA to check if a proper handler for the exception could be found. If the handler is found, libgcc would switch the PC to the handler code. Otherwise, libgcc goes to *Step-3*.
- *Step-3:* The libgcc recovers the registers saved by the current function and removes its stack frame by adjusting the stack pointer with the help of CFI. Until now, the PC is set to the return address and repeats *Step-2*. However, if the stack is empty, libgcc would exit abnormally.

In summary, the CFI is a crucial structure used for unwinding the stack and handling exceptions.

2.2 Call Frame Information

To properly unwind the stack, each function is associated with its corresponding CFI, which is essentially a table that describes how to restore the previous call frame by setting the registers appropriately for every address in the program text [27]. The CFI is encoded using the DWARF standard [45].

CFI defines a virtual address CFA (Canonical Frame Address) which is the address other addresses within the call frame can be relative to. CFA points to the base of stack frame regularly. To represent CFA and other saved registers, call frame instructions are defined in every CFI as shown in Figure 2b. In line 2, `DW_CFA_def_cfa` takes two operands representing a register (`rsp`) and an offset (`16`), which defines the CFA to use the provided register and offset. Thus, CFA is represented as $CFA = rsp + 16$. In line 3, `DW_CFA_advance_loc` takes a constant as the operand which is used to create a new table row with the specified location and all the following rules are recorded in the location. `DW_CFA_offset` takes two operands representing the saved register (`rip`) and the offset relatives to CFA. As shown in line 4, the `rip` is represented as $rip = *(cfa - 8)$. `DW_CFA_def_cfa_offset` takes a constant as the operand which defines current CFA to use the constant as the offset but to keep the predefined register. As shown in line 6, current CFA is represented as $CFA = rsp + 24$. The unwinding table is shown in Figure 2c after interpreting the call frame instructions shown in Figure 2b.

Suppose the program throws an exception at 0x406241 (line 9 in Figure 2a). If the current function cannot handle the exception correctly, the program will unwind the stack. Specifically, the program will search the unwinding table shown in Figure 2c, and find that the current `PC` is within the range of `0x406205` and `0x40625B`. The program will then calculate the `CFA` as `rsp+32` and the `rip` (saved return address) as `*(CFA-8)`. With this information, the program will unwind the stack to the previous caller function.

2.3 Attributes Related to Unwinding

In GCC and Clang compilers, certain function attributes define functions that cannot throw an exception or unwind the stack. Specifically, both GCC [13] and Clang [49] define "nothrow" attributes to inform the compiler that the annotated function does not throw an exception. LLVM defines "nounwind" attribute to indicate that the function never raises an exception [38]. It's worth noting that LLVM would mark some known functions in the standard library, such as `atexit`, `frame_dummy`, and so on, as "nounwind".

2.4 Is Call Frame Information Necessary for Every Function?

The specifications [11, 27] define that every function should have the CFI to support stack unwinding. Pang *et al.* [34] also observed that nearly every function of x64 binaries has the corresponding CFI in real-world applications. However, here comes the question, is the CFI necessary for every function?

Let $G = (V, E)$ be a call graph, where V is the set of nodes (i.e., functions) and E is the set of edges (i.e., calls) in the graph. Let f be a function in G, and let S be the set of successors of f in G. Then, S is the set of functions that can be called from f in G, which means that for any function g in S, there exists an edge between f and g.

Mathematically, this can be expressed as:

$$S = \{g \mid (f, g) \in E\} \tag{1}$$

If one of the successors of f in S unwinds the stack or f raises an exception, then f unwinds the stack[2]. Let $U(f)$ represent whether the function f unwinds the stack or not and $R(f)$ represent whether the function f raises an exception. Mathematically, we could deduce $U(f)$ with the following equation:

$$U(f) = \begin{cases} True, & R(f) = True, \\ \bigvee_{i=1\ldots I} U(g_i) \text{ where: } g_i \in S, & R(f) = False \end{cases} \tag{2}$$

From the above equation, we can infer that if all the functions called by the function f do not unwind the stack and f does not raise an exception, then function f does not unwind the stack either.

2.5 Function Entries Identification

To identify the function entries, the disassemblers present multiple strategies. Mainstream disassemblers first identify the start point (*i.e.,* the `main` function) of the program and perform the recursive disassembling from the start point and add the targets of call instructions as new function entries. However, there still leaves gaps after the disassembling as the existence of indirect calls. To detect the remaining functions, disassemblers scan the non-disassembled code with data-mining models [6, 28] or common function prologues [2, 40]. The above approaches are heuristic and do not guarantee the correctness of function entries [33]. There emerges another way to detect functions that leverages the CFI as the "oracle" of function entries [1, 2, 34]. The related works [33, 34] found that these disassemblers achieve nearly full coverage with high accuracy for x64 binaries.

3 RESEARCH SCOPE

Our objective is to increase the difficulty of disassembling by obfuscating the CFI. Our target architectures are x64 and AArch64 for two reasons: firstly, they are popular and widely supported by most disassemblers. Secondly, their specifications [11, 27] define the CFI of functions. To evaluate the effectiveness of our tool, we have selected disassemblers such as ANGR, GHIDRA, and FETCH that rely on CFI to identify function entries. We have developed the OCFI prototype using Clang/LLVM and believe that it can be easily adapted for other compilers.

4 DESIGN OF OCFI

We present OCFI to obfuscate CFI. Figure 3 shows the overview of OCFI. OCFI contains two major components: no-unwinding propagation and obfuscation. The no-unwinding propagation of OCFI is used to propagate the attribute `nounwind` of known function to other functions among the call graph.

In the obfuscation process, OCFI leverages two strategies to obfuscate the CFI. (1) If the function is marked as `nounwind`, OCFI would set the range of FDE randomly inside the function. (2) If the function may unwind the stack, OCFI would analyze the minimized unwinding range (MUR) that could correctly unwind the stack, and set the range to the MUR.

Algorithm 1: NO-UNWINDING PROPAGATION

```
Input    : Call graph $CG$
/* Compute the strongly connected components of $CG$ */
$SCCs$ = computeSCCs($CG$)
/* bottom-up traversal on SCCs */
for each SCC in SCCs do
    Unwind = False
    for each function F in SCC do
        for each instruction I in F do
            if isa<CallInst>(I) then
                if callee = I.getCalleeFunction() then
                    if MayUnwind(callee) then
                        Unwind = True
                    end
                end
            end
            /* If the instruction is a indirect call, we assume its
               callee(s) unwind the stack conservatively. */
            if isa<IndirectCall>(I) or isa<ResumeInst>(I) then
                Unwind = True
            end
        end
    end
    if ¬ Unwind then
        for each function F in SCC do
            markNounwind(F)
        end
    end
end
```

In the following sections, we will explain in detail the following questions: ❶ how does OCFI propagate `nounwind` attribute among call graph (§4.1); ❷ how does OCFI analyze minimized unwinding range (§4.2); ❸ how does OCFI debloat/obfuscate CFI safely (§4.3).

4.1 No-unwinding Propagation

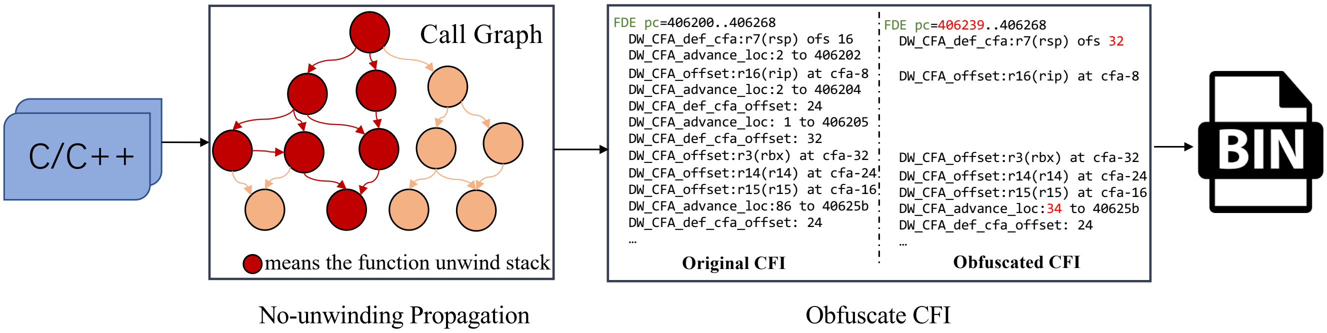
Compilers or programmers mark specific functions as `nounwind`, which represents the functions that would not unwind the stack. To propagate the `nounwind` attribute of the specific functions to other functions, we perform bottom-up analysis on the call graph. As there may exist cycles in the call graph, OCFI groups the call graph into several strongly connected components (SCCs) [48] and leverages bottom-up propagation on the SCCs. The propagation algorithm is shown in algorithm 1.

Specifically, for every SCC, OCFI iterates every function and checks if one of the instructions may unwind the stack, if yes, mark all of the functions inside the SCC as unwinding. We consider the following situations to check if the instruction I may unwind the stack, if one of the situations satisfies, the instruction may unwind the stack at runtime.

(1) If the instruction is a direct call, OCFI extracts the called function and checks if the called function may unwind the stack. As OCFI performs bottom-up iteration on the SCCs,

---

[2] There is a special case where f catches the thrown exception(s) and does not unwind the stack. As we cannot statically determine if f could catch the exception, we conservatively consider that f cannot catch the exception.



```
FDE pc=406200..406268
  DW_CFA_def_cfa:r7(rsp) ofs 16
  DW_CFA_advance_loc:2 to 406202
  DW_CFA_offset:r16(rip) at cfa-8
  DW_CFA_advance_loc:2 to 406204
  DW_CFA_def_cfa_offset: 24
  DW_CFA_advance_loc: 1 to 406205
  DW_CFA_def_cfa_offset: 32
  DW_CFA_offset:r3(rbx) at cfa-32
  DW_CFA_offset:r14(r14) at cfa-24
  DW_CFA_offset:r15(r15) at cfa-16
  DW_CFA_advance_loc:86 to 40625b
  DW_CFA_def_cfa_offset: 24
  …
```

```
FDE pc=406239..406268
  DW_CFA_def_cfa:r7(rsp) ofs 32

  DW_CFA_offset:r16(rip) at cfa-8




  DW_CFA_offset:r3(rbx) at cfa-32
  DW_CFA_offset:r14(r14) at cfa-24
  DW_CFA_offset:r15(r15) at cfa-16
  DW_CFA_advance_loc:34 to 40625b
  DW_CFA_def_cfa_offset: 24
  …
```

Figure 3: The overall workflow of OCFI. The example of Call Frame Instructions (CFI) is based on Figure 2b. In this example, the red color is used to highlight the modifications introduced by OCFI in comparison to the original version.

OCFI could determine the unwinding status of the called function ahead of the caller function.

(2) If the instruction is an indirect call, OCFI could not determine the targets of the indirect call statically, OCFI assumes the indirect call may unwind the stack conservatively.

(3) If the instruction could unwind the stack semantically, such as `ResumeInst` [24] in LLVM IR.

Otherwise, the functions in the SCC should mark as `nounwind`. At the end, OCFI could propagate `nounwind` attribute from known functions to other functions.

4.2 Minimized Unwinding Range

The minimized unwinding range is the minimized range of FDE entry that could be leveraged to unwind the stack correctly. It is leveraged by the function(s) that may unwind the stack. As the FDE entry represents the continuous region that may unwind the stack, the minimized unwinding range is continuous too. OCFI divides functions into two categories based on whether the function is marked as `nounwind`. For the function that is marked as `nounwind`, the minimized unwinding range is empty.

To analyze the minimized unwinding range of the function that may unwind the stack, OCFI iterates the basic blocks of the function linearly. For every basic block BB_i, if one of the instructions may unwind the stack, OCFI deems BB_i unwinds the stack. OCFI could obtain the list of basic blocks that may unwind the stack in the function: $\vec{UB} = \{BB_1, BB_2, ..., BB_n\}$.

OCFI then sorts the $\vec{UB}$ by the address of the basic block and gets the first one F_b and the last one L_b. As OCFI operates the basic block at the end of the optimization, the layout of the basic blocks that OCFI handles is consistent with the layout of binary code. So far OCFI could obtain the minimized unwinding range $[F_b, L_b]$. However, if the F_b is the same as the first basic block of current function, which means that the beginning address of MUR equals the address of function entry. To handle this problem, we make further checks and transformations. Specifically, if the first instruction that may unwind the stack is inside the first basic block of current function, we split the basic block into two basic blocks at the location of first unwinding instruction. By the way, we decided not to always split the first basic block at the first unwinding instruction for the purpose of preserving the original layout of basic blocks as much as possible to avoid affecting subsequent optimizations.

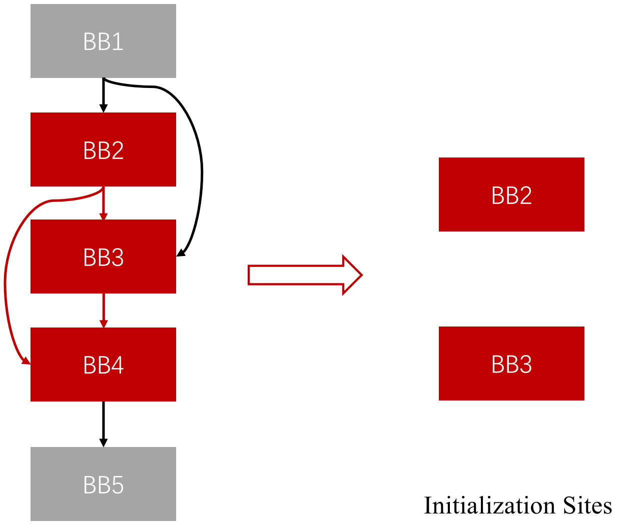


Figure 4: An example to illustrate the determination of initialization sites. A rectangle represents a basic block of a function and a line represents the control flow between basic blocks. The rectangle filled with red represents a basic block inside MUR. As BB2 and BB3 have incoming edges outside MUR, we should initialize registers at the beginning of these basic blocks.

In summary, we could conclude the minimized unwinding range of function f with the following equations:

$$MUR(f) = \begin{cases} \emptyset, & U(f) = True \\ [F_b, L_b], & U(f) = False \end{cases} \tag{3}$$

4.3 Debloat/Obfuscate Call Frame Information

As mentioned in §2, CFI stores register information that could be leveraged to restore the previous frame. To debloat/obfuscate CFI safely, the state of registers inside the minimized unwinding range should be calculated properly. As illustrated in §2, the states of registers represented in the CFI are calculated based on the initialization states. To guarantee the functionality of unwinding the stack, OCFI should satisfy the following two requirements:

- *R1 – The initialization sites inside the minimized unwinding range should be determined.*
- *R2 – The states of registers at initialization sites should be calculated.*

To meet the requirement *R1*, OCFI iterates over every basic block inside the minimized unwinding range and checks if the basic block has the incoming edge(s) whose source is outside the minimized unwinding range. If so, OCFI sets the beginning of the basic block as the initialization site. An example is illustrated in Figure 4.

To meet the requirement *R2*, OCFI calculates the proper states of registers at the initialization sites. There are two kinds of registers that should be considered:

- The canonica frame address (CFA). The CFA is the virtual address that could be leveraged to represent other addresses where callee saved registers are stored.
- The callee saved registers and the return address. These registers should be represented at the unwinding site to restore the previous frame correctly.

OCFI performs data flow analysis on the original CFI to calculate the CFA, the callee saved registers and the return address. The algorithm is shown in algorithm 2. The algorithm is based on the following assumption:

Given a basic block `B`, its immediate predecessors are $PB_1, PB_2, ..., PB_n$. At the end of these predecessor basic blocks, the representation of CFA and callee saved registers are the same. That is, $OUT(PB_1) = OUT(PB_2) = ... = OUT(PB_n)$. Otherwise, the return address and callee-saved registers could not be determined correctly when unwinding occurs inside `B`.

5 IMPLEMENTATION

We implement the prototype of OCFI on LLVM/Clang 12.0. Specifically, to propagate no-unwinding property, we implemented a pass based on `CallGraphSCCPass` [23] which iterates functions on bottom-up orders in strongly connected components of call graph. We calculate the minimized unwinding range during LLVM codegen modular [20]. To calculate the proper CFA and registers of the minimized unwinding range, we reuse the dataflow analysis in `CFIInstrInserter` [22] pass. Lastly, we hook the process of emitting CFI instructions and remove the instructions that out of the minimized unwinding range in `AsmPrinter` [21].

6 EVALUATION

6.1 Dataset

We use two dimensions, availability and effectiveness, to evaluate OCFI. Availability refers to whether the obfuscated binaries can still properly unwind the stack, while effectiveness refers to whether the obfuscated binaries are more challenging for popular disassemblers to identify functions than the original binaries. To test OCFI from these two dimensions, we create the following dataset.

Real-world Software. In order to evaluate the availability and effectiveness of OCFI, We build a large scale dataset of real-world software. The dataset is shown in Table 1. The dataset includes programs and libraries of diverse functionality and complexity, written in C/C++. To test the effect of different compiler options, we built the dataset with various compiler optimizations (O0, O2, O3, Os, Ofast). We built the binaries on two popular architectures (x64 and aarch64). In summary, the dataset contains 1,440 C binaries and 3,034 C++ binaries.

Algorithm 2: CFA & REGISTERS CALCULATION

```
Input  : Function $f$
Input  : CFI instructions of the function $f$: $CFIs$
Output : A list of mappings between basic block to CFA and
         callee saved registers at the beginning of the basic
         block:
         $\vec{BC} = \{(BB_1, cfa_1, \vec{regs}_1), ..., (BB_n, cfa_n, \vec{regs}_n)\}$
Initialization: $\vec{BC} = \emptyset; visited = \emptyset; S = Stack();$
$S$.push(($f$.entry_block(), none, $\emptyset$))
$\vec{BC}$.push(($f$.entry_block(), none, $\emptyset$))
while $\neg S.empty()$ do
    $B, cfa_{ini}, \vec{regs}_{ini} = S$.pop()
    if $visited.contains(B)$ then
        continue
    end
    $visited$.insert($B$)
    $cfa, \vec{regs}$ = $calculate\_cfa\_regs(B, cfa_{ini}, \vec{regs}_{ini})$
    for $each\ BS\ in\ B.successors()$ do
        /* The states at the end of the basic block equal the begin
           of the succeeding basic block(s)                        */
        $S$.push(($BS, cfa, \vec{regs}$))
        $\vec{BC}$.push(($BS, cfa, \vec{regs}$))
    end
end
Procedure calculate_cfa_regs($B, cfa, \vec{regs}_{ini}$):
    /* Iterate over the cfa instruction of the basic block $B$      */
    for $each\ I\ in\ B.CFA\_Instrs()$ do
        /* If the instruction modifies cfa value, update it        */
        if $I.operate\_cfa()$ then
            update_cfa_value($cfa_{ini}, I$)
        end
        if $I.operate\_regs()$ then
            update_regs($regs_{ini}, I$)
        end
    end
return $cfa, \vec{regs}$
```

Automated Generation Programs. In order to generate programs that could trigger unwinding progress, we develop USMITH which could generate C/C++ programs with `try/catch` statements. USMITH is built on Csmith [52], which could generate C programs automatically with predefined rules. Csmith maintains a global environment and a local environment. The global environment holds global scope definitions such as types, global variables and functions. The local environment holds local information about the current generation point, including ① the function call chain information for the current generation point, which is used for context-sensitive pointer analysis, ② the variables that can be referenced by the current generation point, and ③ the alias relationships for local variables. Csmith defines rules for C/C++ code generation, supporting function definitions, global variables, local variables,

```
int depth = 0;
void func1() {
    depth++;
    ... // initialization
    throw 1;
    ...
    depth--;
    return;
}
int main() {
    depth++;
    ... // initialization
    try {
        func1();
    } catch(const int e) {
        printf("Catch the exception, depth is %d!", depth);
    }
    depth--;
}
```

Figure 5: An example of generated binary of USMITH. `try`, `catch`, `throw`, `printf`, and `depth` are inserted by USMITH based on Csmith. `depth` is used to record the depth of calling stack when throwing an exception.

expressions, common control flow (`if/else`, `function call`, `for`, `return`, `break`, `continue`, `goto`), numeric operations, and bit operations. To generate a valid C program, Csmith first generates random types such as structure and global variables, and then defines the main function to generate C statements following top-down generation rules: when a new local variable is defined, the local environment is updated; when a specific type of variable is needed, the appropriate variable is selected from the global or local environment, and the pointer alias relationship is updated. When generating the call site of a function, USMITH randomly decides whether or not to wrap a call in a try/catch and inserts `printf` statement inside the `catch` statement. In the body of some functions, USMITH inserts `throw` statement randomly. An example of the generated binary of USMITH is shown in Figure 5.

To validate the correctness of the obfuscated binaries by OCFI, USMITH compiles the generated C++ program into a normal binary and an obfuscated binary, respectively, and compares the outputs between the two binaries with differential testing. If the outputs are same, we could conclude that the obfuscated binary could catch the threw exception correctly.

6.2 Availability

To test the availability of the obfuscated binaries by OCFI, we performed evaluations on both real-world software and automated generation programs dataset.

Real-world Software. We summarize the real-world software into two categories: ① Spec CPU 2017 and ② C++ applications. For the C++ programs in Spec CPU 2017, we ran the benchmarks automatically and compared the outputs between the obfuscated version and original version. For the C++ applications, we marked the `throw` sites from the source code as the targets and ran directed greybox fuzzing (AFLGO [7]) to generate testcases that reach the `throw` sites automatically. After 24 hours of running, we collected the testcases generated by AFLGO and ran the inputs again to check if the outputs are the same between the obfuscated binary and the original binary.

Table 1: Software used for evaluating tools.

Type	Name	Programs/Binaries	
		C	C++
Benchmark	SPEC CPU2017	32 / 190	24 / 130
Utilities	Findutils-4.4 Binutils-2.26 Coreutils-8.30 Bloaty-1.1 Cppcheck-2.10 Guetzli Libjxl-0.8 Lodepng Matplotlib-cpp Ninja-1.12.0 Poppler-125.0.0 Qpdf-11.0.0 Sentencepiece-0.1.97 Tesseract-5.2.0 Xpdf-4.04 Znc-1.9	242 / 1250	52 / 740
Clients	Alembic-1.8.3 Capnproto-0.11 Grpc-1.49.1 Easywsclient Mosh-1.3.2.95 Protobuf-c-1 Openbabel-3.11 Openthread-0.0.7 PcapPlusPlus-22.05 Rapidxml-1.13 Spicy-1.5.1 Xerces-c-3.2.4	0 / 0	84 / 950
Servers	Mysql-8.0.20 Oatpp-1.3.0 Opendnp3-3.1.2 Wabt-1.0.30	0 / 0	30 / 480
Libraries	Arrow-1000 libsass-1.0.0 bls-signatures-1.0.16 libyuv libzmq-5.25 Nghttp2-14.24.0 Opencv-4.6.0 Pistache-0.0.5 Pugixml-1.12 Re2-10.0.0 Resiprocate-1.13 Snappy-1.9 Spdlog-1.10.0 Spotify-json-3.1.5 Tinygltf2.0 Tinyobjloader-2.0 Tinyxml2-9 Leveldb-1.23 Zopfli-1.0.3	0 / 0	38 / 734
Total		274 / 1,440	228 / 3,034

To check if the program could throw an exception that would unwind the stack, we hooked the `cxa_throw` and `cxa_rethrow` functions of standard C++ library. Specifically, we wrote a library containing customized `cxa_throw` and `cxa_rethrow` which record the number of throw exceptions during running the testcase and redirect the execution to the original functions in the standard C++ library. To hook the functions, we set the path of the customized library to `LD_LIBRARY_PATH`[3].

The real-world dataset we used in the evaluation is shown in Table 2. In the evaluation, we ran 19 C++ applications and 10 of them do not throw exceptions so we omit these applications in table 2. For the remaining 9 applications, there are 4,973 testcases during running or fuzzing which raise 151,773 exceptions. We did not find any difference in immediate outputs between the obfuscated binaries and the original binaries. This indicates that the obfuscated binaries could raise the exception correctly for the real-world dataset shown in Table 2.

---

[3] `LD_LIBRARY_PATH` is an environment that determines where to look for dynamic shared libraries that an application was linked with.

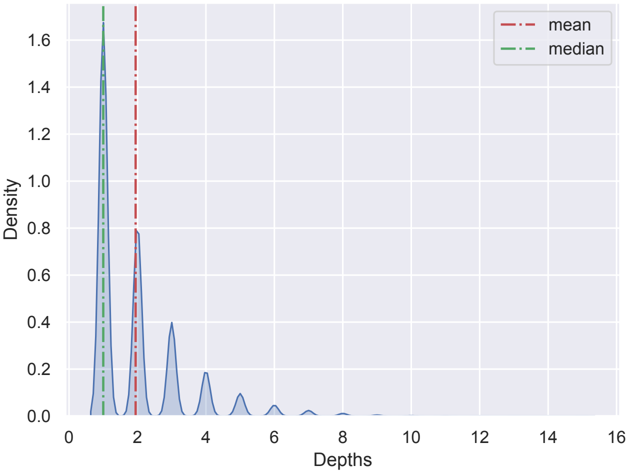


Figure 6: The distributions of stack depths of programs generated by USMITH.

Table 2: Applications used in our evaluation. ✓ indicates the immediate outputs of obfuscated binaries are same as the original binaries. Column `throws` indicates the number of exceptions thrown when running.

Applications			AFL Settings		Results
`Name`	`Category`	`throws`	`Seeds`	`Options`	
omnetpp_r	CPU 2017	10	–	–	✓
omnetpp_s	CPU 2017	10	–	–	✓
leela_r	CPU 2017	10	–	–	✓
leela_s	CPU 2017	10	–	–	✓
parest_r	CPU 2017	139,990	–	–	✓
povray_r	CPU 2017	20	–	–	✓
bloaty	Utilities	1,058	[15]	bloaty @@	✓
qpdf	Utilities	10,599	[29]	qpdf @@ /dev/null	✓
xerces-c	Clients	66	[30]	EnumVal @@	✓

Automated Generation Programs. We ran USMITH to generate C++ programs that contains `try/catch` statements automatically for 605 hours. In summary, we generated 22,323,187 C++ programs that raised 22,323,187 exceptions. To validate that the obfuscated binary could raise the exception correctly at different stack depths, we record the stack depths when raising an exception. Specifically, we declare a global integer variable `depth` and increment the variable when entering a function and decrement the variable when leaving a function. The distributions of stack depths are shown in Figure 6. The mean value of the stack depths is 2 and the median value of the stack depths is 1. Moreover, the max value of the stack depths is 16. It shows that the obfuscated binaries by OCFI could catch the exception properly under different stack depths.

6.3 Effectiveness

To test the effectiveness of OCFI, we evaluated the popular disassemblers' precision ($precision = \frac{|TP|}{|TP|+|FP|}$, TP and FP stand for true positives and false positives) and recall ($recall = \frac{|TP|}{|TP|+|FN|}$, FN stands for false negatives) on function detection by comparing original binaries and obfuscated binaries by OCFI. The dataset is shown in Table 1. The disassemblers we used in the evaluation are shown in Table 3. We extract ground truth of function entries from symbol information.

Table 3: The group of disassemblers that are used in the evaluation.

Tool	*Version*	*Source (Release Date)*
GHIDRA	10.2	Website [1] (Nov 3, 2022)
ANGR	9.2.15	Github [50] (Aug 24, 2022)
FETCH	1.0	Github [53] (Mar 29, 2021)

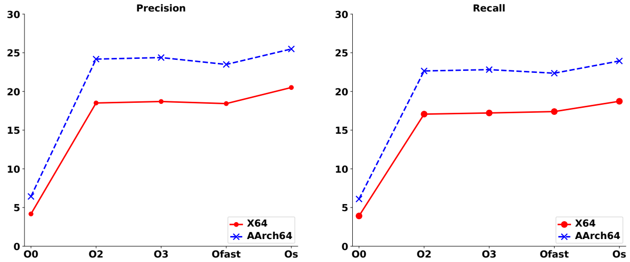
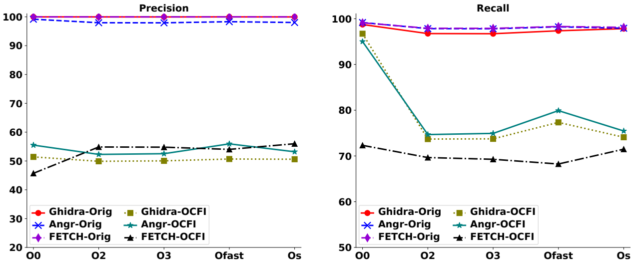
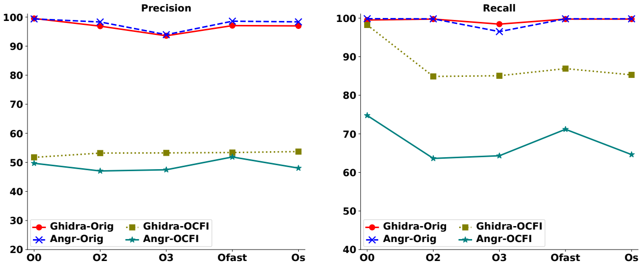


Figure 7: Comparation results between obfuscated call frame information generated by OCFI with symbol information.



(a) X86



(b) AArch64

Figure 8: Evaluation results of function detection of popular disassemblers on the original binaries (with the suffix `Orig`) and the obfuscated binaries by OCFI (with the suffix `OCFI`).

Table 4: Evaluation results of binary size (KB) and size overhead of obfuscated binaries compared with the original binaries (`Orig` in the table). `INC` represents the incremented rate compared between the obfuscated and original binaries (`INC` = $\frac{\texttt{OCFI}-\texttt{Orig}}{\texttt{Orig}} * 100\%$). Avg indicates the average value of the above rows.

OPT	X64						AArch64					
	C			C++			C			C++		
	`Orig`	`OCFI`	`INC`	`Orig`	`OCFI`	`INC`	`Orig`	`OCFI`	`INC`	`Orig`	`OCFI`	`INC`
O0	378.42	378.22	-0.05	1659.02	1679.95	1.26	349.08	345.46	-1.04	1937.60	1927.52	-0.52
O2	270.50	265.56	-1.83	1177.58	1204.87	2.32	254.84	254.34	-0.19	1458.71	1654.65	13.43
O3	284.28	278.75	-1.94	1203.09	1261.29	4.84	263.34	265.52	0.83	1445.53	1682.74	16.41
Os	233.17	228.03	-2.20	1022.05	1050.91	2.82	234.23	233.73	-0.21	1346.77	1458.65	8.31
Of	285.89	280.19	-1.99	1203.99	1255.27	4.26	267.56	267.01	-0.21	1447.89	1683.06	16.24
Avg	290.45	286.15	-1.48	1253.15	1290.51	2.98	273.81	273.21	-0.22	1525.57	1681.32	10.21

We first compared the obfuscated call frame information generated by OCFI with the corresponding symbol information directly. The results of this comparison are presented in Figure 7. We could conclude that OCFI effectively obfuscates the majority of function entries in the call frame information from the results (83.94% in X64 and 79.20% in AArch64). We checked the functions that could not be obfuscated by OCFI and summarized them into the following categories: ① The functions that are statically linked in the executable file. As these functions are not compiled by OCFI, we could not obfuscate them. However, if we compile all the related libraries by OCFI, we could obfuscate them. ② The functions whose first instruction may unwind the stack. For these functions, OCFI would mark the start of the CFI at function start. However, if we leverage accurate alias analysis (such as SVF [46]) to analyze the targets of indirect calls, we could eliminate the assumption about all of the indirect calls may unwind the stack. These functions could be reduced accordingly.

In Figure 8, we conducted an evaluation of popular disassemblers' function detection capabilities on both the original binaries and the obfuscated binaries generated by OCFI. The evaluation results demonstrate that OCFI effectively enhances the difficulty of disassemblers in detecting function entries. More specific, The precision and recall of Ghidra (Precision: 98.40% → 51.75%, Recall: 98.44% → 83.45%), angr (Precision: 97.99% → 51.34% ,Recall: 98.66% → 73.88%), and fetch (Precision: 99.99% → 53.05%, Recall: 98.26% → 70.19%) reduce largely on x64 and AArch64.

6.4 Size Overhead

The `eh_frame` section is loaded to the address space of a program when loading and it could not be stripped. We evaluated the size overhead of obfuscated binaries in Table 4.

In summary, the average size overhead of the obfuscated binaries is 4%. Specifically, the size of C obfuscated binaries is slightly smaller than the original binaries (x64: -1.48%, AArch64: -0.22%). We found that most functions of C do not unwind the stack and OCFI removes some of the call frame instructions randomly. The size of C++ obfuscated binaries is slightly larger than the original binaries (x64: 2.98%, AArch64: 10.21%). We found that some of the obfuscated C++ functions have many initialization sites for the states of registers, which would increase the size of `.eh_frame`.

Table 5: Runtime overhead under Spec CPU2017 benchmarks. Columns `Orig` and `OCFI` list the running time (seconds) of original binaries and obfuscated binaries. `INC` represents the incremented rate compared between the obfuscated and original binaries (`INC` = $\frac{\texttt{OCFI}-\texttt{Orig}}{\texttt{Orig}} * 100\%$). Avg indicates the average value of the above rows.

Program	C/C++	X64			AArch64 (QEMU)		
		`Orig`	`OCFI`	`INC`	`Orig`	`OCFI`	`INC`
500.perlbench_r	C	248.1	247.2	-0.4	1728.2	1730.9	0.2
502.gcc_r	C	181.4	180.8	-0.3	977.4	977.8	0.0
505.mcf_r	C	247.8	248.1	0.1	875.8	872.8	-0.3
520.omnetpp_r	C++	330.2	333.5	1.0	1244.3	1259.7	1.2
523.xalancbmk_r	C++	252.5	252.2	-0.1	1033.2	1054.7	2.1
525.x264_r	C	195.9	197.6	0.9	1510.8	1508.4	-0.2
531.deepsjeng_r	C++	211.9	211.0	-0.4	1072.2	1078.0	0.5
541.leela_r	C++	346.7	349.2	0.7	1722.3	1719.5	-0.2
557.xz_r	C	276.9	278.6	0.6	862.0	862.8	0.1
508.namd_r	C++	192.4	192.5	0.1	4045.3	4017.6	-0.7
510.parest_r	C++	349.9	350.3	0.1	4343.9	4316.6	-0.6
511.povray_r	C, C++	297.9	295.0	-1.0	5140.4	5133.3	-0.1
519.lbm_r	C	176.7	176.8	0.1	3237.8	3289.7	1.6
526.blender_r	C, C++	222.9	222.8	-0.1	2699.8	2713.5	0.5
538.imagick_r	C	345.5	346.8	0.4	7473.0	7515.3	0.6
544.nab_r	C	262.1	262.9	0.3	5891.1	5902.3	0.2
Avg	–	258.7	259.1	0.1	2741.1	2747.1	0.3

6.5 Runtime Overhead

To test the runtime overhead, we compiled the Spec CPU2017 with O2 optimization level and run the obfuscated and original binaries. The machine we used for the evaluation is Intel i7-10700K CPU 3.80 GHz, Ubuntu 20.04. For AArch64, we evaluate under Qemu

Table 6: Distributions of function entries identification. `eh_frame` indicates that the function entries are correctly marked in the `.eh_frame`, `call` indicates the functions are recognized by direct call and `matching` indicates the functions are found by function pattern matching.

	eh_frame	matching	call
Ghidra	21.86%	17.56%	60.58%
angr	25.84%	9.42%	64.74%

emulation. Specifically, we ran the evaluations 5 times and filtered the largest and smallest value and calculated the average value of the remaining values. The results are shown in Table 5. The results show that ocfi incurs nearly zero runtime overhead (0.2% on average) compared to the original binaries.

7 DISCUSSION

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

7.1 Threats of Validity

In this study, we have concentrated on obfuscating CFI with the aim of misleading the function detection results of popular disassemblers. Disassemblers may leverage dataflow analysis to detect the existence of obfuscated CFI. For example, FETCH [34] checks the validity of calling conventions before marking function entries from `.eh_frame`. However, our experiments show that the precision of FETCH on x64 platform is only 53.05% (shown in Figure 8a). This indicates that this approach is not easily scalable enough to detect obfuscated CFI.

Additionally, our study, as well as that of Pang *et al.* [33], has discovered that some disassemblers (e.g., Ghidra and angr) perform poorly without CFI, which suggests that ocfi can also be used to prevent disassemblers from identifying function entries with the aid of the CFI section and can increase the difficulty of detecting function entries.

By the way, there are some possibilities to deobfuscate ocfi. One possible deobfuscation is searching function prelude patterns among the gaps left by the ranges of CFI that Ghidra and angr already leverage this strategy. There are three sources used to identify function entries of Ghidra and angr: ① the `.eh_frame`, ② direct calls, and ③ function pattern matching. We show the distributions of these three sources on truly identified functions recognized by Ghidra and angr on the ocfi obfuscated binaries in Table 6. The results show that the function pattern matching among the gaps left by the ranges of CFI only recovers few function entries.

7.2 Future Works

While ocfi conservatively assumes that the targets of indirect calls unwind the stack, it may generate false negatives when propagating the "`nounwind`" attribute. To address this issue, we plan to use precise alias analysis of pointers to determine the targets of indirect calls in future work.

ocfi marks the beginning of the minimized unwinding range at the start of a specific basic block, which gives a clue about identifying correct instructions. As shown in Table 6, direct calls contribute significantly to the majority of identified function entries. In the future, we plan to mark the beginning of the minimized unwinding range at the address of some incorrect instructions, which could potentially mislead disassemblers into identifying the wrong direct call instructions.

8 RELATED WORKS

Obfuscating binaries. Software obfuscation aims to make programs harder to understand or analyze, without impacting expected functionality. Over the years, various obfuscation methods [9, 10, 18, 19, 51, 54] were proposed, and combinations of these techniques [42, 43] were also been extensively studied for more effective confusion. The most related category to our work is static code rewriting, including data obfuscation and control flow obfuscation. Data obfuscation such as Mixed Boolean Arithmetic [42, 54] is suitable for limited scenarios because of the unbearable overhead in binary transformation or encrypt. The need of the target determines the obfuscation level since high obfuscation increases cost [16]. Thus we propose a novel way based on the mechanism of stack unwinding to fit that specific target well. Control flow obfuscation is a popular way to stop higher-level abstractions recovery from assembly, including bogus insertion [10], opaque predicates [10, 51, 55], and control flow flattening [9, 19]. Balachandran *et al.* [4] removes code blocks randomly to a new code segment with order-preserving by jump instruction in order to disturb control flow. ROPOB [31] conceals control flow between CFG basic blocks utilizing Return Oriented Programming (ROP). Schrittwieser *et al.* [43] splits assembly code into small pieces and combines them by branching function, and it also inserts dummy code to improve strength. However, these works also show obvious costs due to program complexity increasing, instruction modification, code instrumentation, etc. For instance, [4] brings 1.25x average time overhead while the file size obfuscated with ROPOB [31] expands to 2.66x in average. What's more, there's no obfuscation on CFI since all studies mentioned above focus on control flow and work on assembly code level, including ROPOB which takes binary code as input. In other words, our work is complementary to control flow obfuscations.

The key to proof availability of obfuscation is how to definite "functionally equivalent", since living with the cost (e.g., program size or time) is a consensus. Collberg *et al.* [10] considers the relationship between the original and the obfuscated program to be "weakly equivalent", with only one requirement that the observable behavior (the behavior experienced by the user) of the two programs should be the same. Similarly, the definition of "harder to understand and analyze" is under discussion. Some compiler optimizations are also considered obfuscation, because code that improves performance may do the opposite in understanding [5], which affects disassembly subtly.

Call Frame Information. The specifications [11, 27] define that each function should have CFI for stack unwinding and exception handling, which gives reverse engineering a chance. Oakley *et al.* [32] hides malicious instructions in DWARF-format CFI to gain the control flow of execution. Duta *et al.* [12] leverages corrupted stack unwinding path to hijack control flow. Some researches [34, 36] note the role of .eh_frame in function identification. While mainstream binary analysis tools [1, 2] also use CFI to detect function starts. Priyadarshan *et al.* [36, 37] points out that when .eh_frame exists, code randomization which aims to resist code reuse attacks shows a decline in performance. They propose mitigation that randomizes call-containing and nearby unwinding information. However, they assume the attacker is experienced and skilled, which is very different from fighting against disassembly in minimal time and space overhead.

9 CONCLUSION

We introduce OCFI, a prototype for obfuscating CFI to make it harder for popular disassemblers to detect function entries. The main idea behind OCFI is that not every function or instruction unwinds the stack at runtime. First, OCFI propagates the nounwind attributes of known functions to other functions and determines the minimized unwinding range of unwinding stacks. Then, OCFI marks the range of CFI as the minimized unwinding range. To evaluate OCFI, we built a large-scale dataset consisting of real-world software and automated generation programs. Our evaluations show that the obfuscated binaries generated by OCFI are more difficult for popular disassemblers to detect function entries. Additionally, OCFI incurs acceptable size overhead (4% on average) and runtime overhead (0.2% on average).

ACKNOWLEDGMENTS

We would thank the anonymous reviewers for their feedback. This work was supported, in part, by grants from the Chinese National Key R&D Program (2022YFF0604503) and the Chinese National Natural Science Foundation (62032010, 62172201). 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] National Security Agency. 2019. *Ghidra Software Reverse Engineering Framework.* https://ghidra-sre.org/ Accessed Dec 2, 2022.

[2] National Security Agency. 2022. *Angr: A powerful and user-friendly binary analysis platform!* https://github.com/angr/angr Accessed Dec 2, 2022.

[3] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, 177–189. https://doi.org/10.1109/EuroSP.2017.11

[4] Vivek Balachandran and Sabu Emmanuel. 2013. Software protection with obfuscation and encryption. In *Information Security Practice and Experience: 9th International Conference, ISPEC 2013, Lanzhou, China, May 12-14, 2013. Proceedings 9.* Springer, 309–320. https://doi.org/10.1007/978-3-642-38033-4_22

[5] Sebastian Banescu and Alexander Pretschner. 2018. A tutorial on software obfuscation. *Advances in Computers* 108 (2018), 283–353. https://doi.org/10.1016/bs.adcom.2017.09.004

[6] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14).* 845–860.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2329–2344. https://doi.org/10.1145/3133956.3134020

[8] Nicholas Carlini and David Wagner. 2014. {ROP} is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14).* 385–399.

[9] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A Zakharov. 2001. An approach to the obfuscation of control-flow of sequential computer programs. In *Information Security: 4th International Conference, ISC 2001 Malaga, Spain, October 1–3, 2001 Proceedings 4.* Springer, 144–155. https://doi.org/10.1007/3-540-45439-X_10

[10] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations.* Technical Report. Department of Computer Science, The University of Auckland, New Zealand.

[11] Arm Developer. 2022. *DWARF for the Arm® 64-bit Architecture (AArch64).* https://github.com/ARM-software/abi-aa/blob/main/aadwarf64/aadwarf64.rst Accessed Dec 10, 2022.

[12] Victor Duta, Fabian Freyer, Fabio Pagani, Marius Muench, and Cristiano Giuffrida. 2023. Let Me Unwind That For You: Exceptions to Backward-Edge Protection. In *NDSS.*

[13] GCC GNU. 2022. *Declaring Attributes of Functions.* https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html Accessed Dec 10, 2022.

[14] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 575–589.

[15] Joshua Haberman. 2017. *Testcases of ELF.* https://github.com/google/bloaty/tree/main/tests/testdata/linux-x86_64

[16] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. 2018. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology* 104 (2018), 72–93. https://doi.org/10.1016/j.infsof.2018.07.007

[17] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP).* IEEE, 969–986. https://doi.org/10.1109/SP.2016.62

[18] Johannes Kinder. 2012. Towards static analysis of virtualization-obfuscated binaries. In *2012 19th Working Conference on Reverse Engineering.* IEEE, 61–70. https://doi.org/10.1109/WCRE.2012.16

[19] Tımea László and Ákos Kiss. 2009. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30, 1 (2009), 3–19.

[20] LLVM. 2017. *Modular Codegen.* https://llvm.org/devmtg/2017-10/slides/Blaikie-Modular%20Codegen.pdf Accessed Jan 1, 2023.

[21] LLVM. 2020. *Source code of AsmPrinter.* https://github.com/llvm/llvm-project/blob/llvmorg-12.0.0/llvm/lib/CodeGen/AsmPrinter/AsmPrinter.cpp Accessed Jan 3, 2023.

[22] LLVM. 2020. *Source code of CFIInstrInserter.* https://github.com/llvm/llvm-project/blob/llvmorg-12.0.0/llvm/lib/CodeGen/CFIInstrInserter.cpp Accessed Jan 3, 2023.

[23] LLVM. 2022. *llvm::CallGraphSCCPass Class Reference.* https://llvm.org/doxygen/classllvm_1_1CallGraphSCCPass.html Accessed Dec 6, 2022.

[24] LLVM. 2023. *llvm::ResumeInst Class Reference.* https://llvm.org/doxygen/classllvm_1_1ResumeInst.html Accessed Jan 5, 2023.

[25] H.j. Lu, David L Kreitzer, Millind Girkar, and Zia Ansari. 2015. *System V Application Binary Interface, Intel386 Architecture Processor Supplement.* https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/intel386-psABI-1.1.pdf Accessed Dec 2, 2022.

[26] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 389–400. https://doi.org/10.1109/TSE.2017.2655046

[27] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2012. *System V Application Binary Interface, AMD64 Architecture Processor Supplement.* https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf Accessed Dec 2, 2022.

[28] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* 24–35. https://doi.org/10.1145/2931037.2931047

[29] Max Moroz. 2019. *Testcases of PDF.* https://github.com/google/AFL/blob/master/testcases/others/pdf/small.pdf

[30] Max Moroz. 2019. *Testcases of XML.* https://github.com/google/AFL/blob/master/testcases/others/xml/small_document.xml

[31] Dongliang Mu, Jia Guo, Wenbiao Ding, Zhilong Wang, Bing Mao, and Lei Shi. 2018. ROPOB: obfuscating binary code via return oriented programming. In *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22–25, 2017, Proceedings 13.* Springer, 721–737. https://doi.org/10.1007/978-3-319-78813-5_38

[32] James Oakley. 2011. Exploiting the {Hard-Working}{DWARF}: Trojan and Exploit Techniques with No Native Executable Code. In *5th USENIX Workshop on Offensive Technologies (WOOT 11).*

[33] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 833–851. https://doi.org/10.1109/SP40001.2021.00012

[34] Chengbin Pang, Ruotong Yu, Dongpeng Xu, Eric Koskinen, Georgios Portokalidis, and Jun Xu. 2021. Towards Optimal Use of Exception Handling Information for Function Detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 338–349. https://doi.org/10.1109/DSN48987.2021.00046

[35] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. 2022. Ground Truth for Binary Disassembly is Not Easy. In *31st USENIX Security Symposium (USENIX Security 22)*. 2479–2495.

[36] Soumyakant Priyadarshan, Huan Nguyen, and R Sekar. 2020. On the impact of exception handling compatibility on binary instrumentation. In *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation*. 23–28. https://doi.org/10.1145/3411502.3418428

[37] Soumyakant Priyadarshan, Huan Nguyen, and R Sekar. 2020. Practical fine-grained binary code randomization. In *Annual Computer Security Applications Conference*. 401–414. https://doi.org/10.1145/3427228.3427292

[38] LLVM Project. 2022. *LLVM Language Reference Manual*. https://llvm.org/docs/LangRef.html Accessed Dec 10, 2022.

[39] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. 2012. Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security* 12, 8 (2012), 74–83.

[40] radreorg. 2020. *Radare2 Github Repo*. https://github.com/radareorg/radare2/tree/5a1df188

[41] Yutaka Sasaki et al. 2007. The truth of the f-measure. 2007. *URL: https://www.cs. odu. edu/mukka/cs795sum09dm/Lecturenotes/Day3/F-measure-YS-26Oct07. pdf [accessed 2021-05-26]* 49 (2007).

[42] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. 3055–3073.

[43] Sebastian Schrittwieser and Stefan Katzenbeisser. 2011. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding: 13th International Conference, IH 2011, Prague, Czech Republic, May 18-20, 2011, Revised Selected Papers 13*. Springer, 270–284. https://doi.org/10.1007/978-3-642-24178-9_19

[44] Hovav Shacham, E Buchanan, R Roemer, and S Savage. 2008. Return-oriented programming: Exploits without code injection. *Black Hat USA Briefings (August 2008)* (2008).

[45] Dwarf std. 2010. *DWARF Debugging Information Format, Version 4*. https://dwarfstd.org/doc/DWARF4.pdf Accessed Dec 6, 2022.

[46] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266. https://doi.org/10.1145/2892208.2892235

[47] Rabia Tahir. 2018. A study on malware and malware detection techniques. *International Journal of Education and Management Engineering* 8, 2 (2018), 20. https://doi.org/10.5815/ijeme.2018.02.03

[48] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160. https://doi.org/10.1109/SWAT.1971.10

[49] Clang Team. 2022. *Attributes in Clang*. https://clang.llvm.org/docs/AttributeReference.html Accessed Dec 10, 2022.

[50] SEFCOM at Arizona State University the Computer Security Lab at UC Santa Barbara. 2022. *Angr: A powerful and user-friendly binary analysis platform!* https://github.com/angr/angr/tree/v9.2.15

[51] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2016. Generalized dynamic opaque predicates: A new control flow obfuscation method. In *Information Security: 19th International Conference, ISC 2016, Honolulu, HI, USA, September 3-6, 2016. Proceedings 19*. Springer, 323–342. https://doi.org/10.1007/978-3-319-45871-7_20

[52] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. https://doi.org/10.1145/1993316.1993532

[53] Ruotong Yu. 2022. *FETCH: A fast and easy-to-use tool to find function entries from x86/x64 System-V binaries (stripped or not)*. https://github.com/ruotongyu/FETCH

[54] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. 2007. Information hiding in software with mixed boolean-arithmetic transforms. In *Information Security Applications: 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers 8*. Springer, 61–75. https://doi.org/10.1007/978-3-540-77535-5_5

[55] Lukas Zobernig, Steven D Galbraith, and Giovanni Russello. 2019. When are opaque predicates useful?. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 168–175.

Received 2023-02-16; accepted 2023-05-03