How’d Security Benefit Reverse Engineers? 

The Implication of Intel CET on Function Identification 

Hyungseok Kim*,†, Junoh Lee†, Soomin Kim†, SeungII Jung†, Sang Kil Cha†
*The Affiliated Institute of ETRI, †KAIST, ‡KAIST CSRC

hskim@nsr.re.kr, junoh@kaist.ac.kr, soomin@kaist.ac.kr, sijung@kaist.ac.kr, sangkile@kaist.ac.kr

Abstract—As CPU vendors introduce various hardware-assisted security features, modern compilers have started to produce binaries containing security-related instructions. Interestingly, such instructions tend to alter the shape of resulting binaries, which can potentially affect the effectiveness of binary analysis. This paper presents the first systematic study on the implication of the Intel CET (Control-flow Enforcement Technology) instructions on function identification. Our study finds that CET-relevant instructions provide useful, although limited, hints for function entries. Therefore, we devise a novel function identification algorithm that utilizes the usage patterns of CET instructions, and demonstrate a tool named FunSeeker that implements the idea. Our evaluation shows that FunSeeker significantly outperforms current state-of-the-art function identification tools in terms of both correctness and speed.

Index Terms—function identification, binary analysis, Intel CET, reverse engineering

I. INTRODUCTION

Memory corruption bugs have existed for about a half century. Yet, they pose a significant security threat, allowing an attacker to take full control over the victim machines.

Although there have been a variety of research attempts to mitigate memory-based exploits [1], [10], [18], [25], [27], [34], [34], [47], those techniques suffer from significant performance overhead, making them difficult to be used in practice.

Hardware vendors attempt to address the performance challenge by introducing various security features as well as new instruction sets. For example, Intel CET (Control-flow Enforcement Technology) [22], which is designed to enforce the Control-Flow Integrity (CFI) [1] at a hardware level, employs a set of new instructions to protect indirect branches with negligible performance overhead [45]. In particular, potential jump targets are marked with a special instruction (ENDBR32 or ENDBR64), indicating a valid destination of an indirect jump. We call such a marker instruction as end-branchn instruction. ARM (ARMv8-A) also supports CFI by introducing an end-branchn instruction named BTI (Branch Target Identification) [5], which behaves similarly to Intel’s ENDBR32 (or ENDBR64).

Recently, major compilers, such as GCC and Clang, have been adapted to emit such end-branchn instructions by default. In an ideal scenario, compilers should place an end-branchn instruction only at a target of an indirect branch. In practice, however, compilers often regard most function entries as a potential jump target as it is difficult to statically resolve all indirect jump targets.

Thus, the natural questions arise: Can an end-branchn instruction exist at a program point other than a function entry? How

II. INTEL CET BACKGROUND

Intel Control-flow Enforcement Technology (CET) is designed to enforce Control-Flow Integrity (CFI) at a hardware level. Particularly, Intel CET includes two different memory protection mechanisms—Shadow Stack (SS) and Indirect
Branch Tracking (IBT)—to protect forward and backward indirect control flows, respectively.

First, SS is a hardware-based shadow stack implementation, which aims to store redundant copies of return addresses to protect backward indirect control flows, i.e., return edges with ret instructions. Those shadow copies can be used to detect stack-based buffer overflows [1], [36].

Second, IBT protects forward indirect branches, such as jmp and call instructions. IBT checks for every indirect branch instruction if it jumps to a pre-defined code location marked via an end-branch (ENDBR32 or ENDBR64) instruction. Figure 1 shows (a) an example C program containing a switch case statement, and (b) the CET-enabled x86-64 binary counterpart. Note that every function in the binary starts with ENDBR64, which indicates that every function can potentially be a jump target of an indirect branch. In Line 12 of Figure 1b, there is an indirect jump prefixed with NOTRACK. The instruction represents the switch statement in Figure 1a, and the prefix is used to mean that the instruction does not need to advance to an end-branch instruction because compilers typically put an input range checking before the indirect jump instruction. Thus, compilers do not insert an end-branch instruction for switch-case clauses [28].

In this paper, we are mainly interested in the IBT feature of CET, which utilizes end-branche instructions. We note that major compilers provide a command-line option (–fcf-protection) to control the level of security to be enforced by CET. By default, compilers turn on the full protection (i.e., –fcf-protection=full) even though we do not give any relevant option [15]. Therefore, in the rest of the paper, when we say a CET-enabled binary, it means the binary is compiled with both SS and IBT features turned on.

III. ANALYSIS OF END-BRANCH INSTRUCTIONS

To study the impact of end-branche instructions, we first collect CET-enabled binaries by compiling real-world packages including 108 programs in GNU Coreutils (v9.0), 15 programs in GNU Binutils (v2.37), and 47 programs in SPEC CPU 2017 benchmark (§III-A). We then linearly disassemble every binary in our dataset to understand the usage patterns of end-branche instructions. Specifically, we analyze where each end-branche instruction is located to see if it can be placed at another place other than a function entry (§III-B). Next, we examine the syntactic properties of all the functions, which may or may not include an end-branche instruction (§III-C). Finally, we summarize our findings and discuss the implication of end-branche instructions in terms of function identification (§III-D).

A. Our Dataset

We used two major compilers (GCC and Clang) to produce our dataset with varying compiler flags, architectures, optimization levels in order to obtain a diverse set of CET-enabled binaries. Since both GCC (v10.0) and Clang (v13.0) emit CET-enabled binaries by default, we did not have to specify the –fcf-protection option. We consider both Position-Independent Executables (PIEs) and non-PIEs because they often result in significantly different shapes. We target two different Intel architectures (x86 and x86-64), and six different optimization levels (00, 01, 02, 03, Os, and Ofast). This gives us 24 (= 2 × 2 × 6) different configurations per binary. We obtained a total of 8,136 CET-enabled binaries. Note that Clang did not produce setbuf binary from GNU Coreutils due to a package configuration error.

All the binaries were compiled with debugging information enabled (with the –g option) in order to extract precise ground truths. However, we stripped all the binaries when evaluating function identification algorithms. We publicize both original and stripped binary datasets.

B. End-Branch Locations

We first examined our dataset to find out where the end-branche instructions are. As a result, we found three different locations: (1) at a function entry, (2) after an indirect-return function call, and (3) at an exception catch block. Table I shows the distribution of end-branche instruction locations in each different set of binaries in our dataset. While the majority of end-branche instructions were found at a function entry, more than 20% of the cases were found in an exception handling (catch) block for the SPEC binaries. Note that the SPEC benchmark includes C++ programs, while Coreutils and Binutils do not. This result highlights that simply regarding an end-branche instruction as a function entry would produce a considerable amount of false positives for C++ binaries.

1) End-Branch at a Function Entry: Compilers tend to add an end-branche instruction at every non-static function entry because one cannot decide whether a non-static function
be referenced by a function pointer before linking. Indeed, most of the end-branch instructions in our dataset are located at a function entry. However, this does not mean that every function starts with an end-branch instruction. For example, static functions do not have an end-branch instruction unless they are referenced by a function pointer. We also found that there are many other cases where functions do not include an end-branch instruction (see §III-C).

2) *End-Branch at an Indirect-Return Function Call*: Functions may return via an indirect jump instruction instead of a ret. Such functions have the indirect-return attribute [13], and an end-branch instruction is inserted right after the call site [12], [14] to protect the return edge. For example, *setjmp* is used to save the current execution context in a dedicated buffer, and the context can be restored by *longjmp*. Figure 2a illustrates this case, where an end-branch instruction is placed right after the call instruction at 0x40a9f9. At the end of the *longjmp* call, the indirect jump instruction at 0x40a9fe will transfer the control flow of the program to 0x40a9fe. We also found that compilers predefined a list of indirect-return functions, such as *setjmp*, *sigsetjmp*, and *vfork* [11]. Therefore, we can easily decide whether an end-branch instruction is for handling an indirect-return function call or not.

3) *End-Branch at an Exception Handling Block*: C++ exceptions are handled by the *libstdc++* library, which uses an indirect jump to transfer the control to a catch clause. Therefore, each catch clause starts with an end-branch instruction. In our dataset, C++ binaries from SPEC CPU 2017 show end-branch instructions at an exception handling block. Figure 2b presents an example of catch block located at 0x109821 of the 508.namd binary.

Fig. 2: Usage patterns of end-branch instructions.

(a) *setjmp* example from *ls* (Coreutils).

(b) Exception handling example from 508.namd (SPEC CPU 2017).

2017 show end-branch instructions at an exception handling block. Figure 2b presents an example of catch block located at 0x109821 of the 508.namd binary.

C. Functions without an End-Branch Instruction

Now that we know end-branch instructions can be placed at several different places other than a function entry, we now study how many functions indeed start with an end-branch instruction. To this end, we first extracted 11,209,121 functions from our dataset using the debugging symbols. We then linearly disassembled the entire code section of each binary to check if the following three properties hold for each of the functions:

- **EndBrAtHead**: there is an end-branch at the entry.
- **DirJmpTarget**: there is a direct jump to the function.
- **DirCallTarget**: there is a direct call to the function.

Figure 3 illustrates how each of the properties holds and how they overlap with each other. Note that about 89.3% \(( \approx 48.85 + 37.79 + 1.44 + 1.23 )\) of the functions start with an end-branch instruction (**EndBrAtHead**). This means one cannot completely retrieve function entries by simply looking at end-branch instructions.

For the 11% of the functions without an end-branch instruction, we found most of them are referenced by a direct jump or a direct call instruction. In other words, we found that at least one of the three properties holds for 99.99% of the functions. The last two properties (**DirJmpTarget** and **DirCallTarget**) are useful to make up for the lack of end-branch instructions because one can easily obtain the target address of direct branches. We further analyzed the rest 0.01% of the functions that do not satisfy any of the properties, and found that they are all dead code that is never referenced by any other instruction.

D. Implication of End-Branch Instructions

Recall from §III-B and §III-C, end-branch instructions have a limited application in detecting functions: (1) it can be placed to non-function locations, and (2) not every function has an end-branch instruction.
Algorithm 1: FunSeeker overview.

1. function FunSeeker (bin)
2. txt, exn ← PARSE(bin)
3. \( E, C, J \leftarrow \text{DISASSEMBLE}(\text{txt}) \) // §IV-B
4. \( E' \leftarrow \text{FILTERENDBR}(E, \text{exn}) \) // §IV-C
5. \( J' \leftarrow \text{SELECTTAILCALL}(J) \) // §IV-D
6. return \( E' \cup C \cup J' \)

However, our study also reveals that there are potentially exploitable patterns for both cases. First, there are only two possible locations where end-branch instructions can be placed other than a function entry, and both the locations are a callee of a special function, such as `longjmp`. Therefore, the challenge is in distinguishing whether an end-branch instruction is a call target of such a special function (§IV-C). Second, most of the functions without an end-branch instruction are referenced by one or more direct branch instruction. Therefore, we should be able to identify those functions by analyzing targets of direct branches. The key challenge is to determine which of the targets represents a function entry (§IV-D).

IV. SYSTEM DESIGN

This section introduces FunSeeker, a lightweight and efficient function identification tool. FunSeeker leverages syntactic patterns we found from CET-enabled binaries (§III) to achieve efficient function identification.

A. Main Algorithm

Algorithm 1 presents the steps taken by FunSeeker to identify functions from a given binary. At a high level, FunSeeker takes in a binary bin as input, and returns a set of function start addresses as output.

PARSE (in Line 2) first analyzes the given binary bin to extract the .text section (txt) and the C++ exception information (exn) from it. Note exn only exists for C++ binaries, and thus, it does not affect C binaries. Next, DISASSEMBLE (in Line 3) linearly disassembles .text, and examines every direct branch instruction to return a 3-tuple \((E, C, J)\), where \(E\) is a set of end-branch instruction addresses found in .text, and \(C\) and \(J\) are a set of direct call targets and a set of direct jump targets, respectively.

FILTERENDBR (in Line 4) then tries to remove end-branch instructions that are not relevant to a function entry to obtain \(E'\) (§IV-C). Next, SELECTTAILCALL (in Line 5) identifies tail calls from a set of direct jump targets \(J\) to get \(J'\) (§IV-D). Finally, Line 6 combines \(E', C\), and \(J'\) to get the final set of function addresses.

B. Disassembly

DISASSEMBLE performs traditional linear-sweep disassembly from the start address of the given .text section txt until reaching the end of the section. In case there is a disassembly error, we increase the program counter by one, and resume the disassembly process. The primary goal here is to find all the end-branch instructions as well as direct jump instructions. Linear-sweep disassembly is known to be effective in recovering instructions from regular binaries generated by a compiler, especially because both major compilers (GCC and Clang) for x86 and x86-64 do not insert data inside the .text section [3], [32]. Although linear-sweep disassembly suits our needs, it may cause errors when dealing with binaries that contain data within a .text section, e.g., hand-written assembly code. Distinguishing between code and data in an arbitrary binary is a well-known undecidable challenge, and handling the issue is beyond the scope of this paper.

C. Filtering out End-Branch Instructions

Recall from §III-B, end-branch instructions can be placed at two different locations other than a function entry. To reduce false-positives in identifying functions, we need to filter out end-branch instructions that are placed either after an indirect-return function call or at an exception catch block.

First, FILTERENDBR checks for every end-branch instruction to see if there is a preceding call instruction that refers to a Procedure Linkage Table (PLT) entry. If so, it retrieves the target function name and compares it with a list of predefined indirect-return function names. Specifically, we use a total of five known indirect-return functions defined by GCC [11], which includes `setjmp` and `vfork`. When the target function name matches with one from the list, FILTERENDBR simply removes the corresponding end-branch instruction address from \(E\) because it represents a return target of the indirect-return function.

Second, FILTERENDBR analyzes every Language-Specific Data Area (LSDA) of the .gcc_except_table section to see if there is any end-branch instruction that belongs to a landing pad, i.e., an exception catch block. The .gcc_except_table section is essential in handling C++ exceptions, and cannot be stripped.

Our approach is orthogonal to the one of FETCH [33], which uses the PC begin values in a Frame Description Entry (FDE) located at the .eh_frame section. Strictly speaking, the FDE records are not essential for x86 binaries as long as they do not include exception handling code. For this reason, Clang indeed does not create an FDE record for every function when the x86 binary is purely written in C, which makes FETCH suffers in dealing with x86 binaries compiled with Clang. However, FunSeeker uses LSDAs stored in the .gcc_except_table section, which cannot be stripped off. Thus, our system is robust against varying compilers.

D. Tail Call Selection

SELECTTAILCALL selects a subset of \(J\) because not every direct jump target represents a function: A direct jump target is a function only when the jump is a tail call. Therefore, existing binary analysis frameworks employ several heuristics to detect tail calls [32].

FunSeeker regards a direct jump as a tail call when the following two conditions are met: (1) if the target address is beyond the boundary of the current function that the jump instruction belongs to, and (2) if the target address
is referenced by multiple functions other than the current function. The first condition is suggested by Qiao et al. [39] and the second condition is inspired by FETCH [33]. Note that checking both conditions does not require any complex analysis technique. The main benefit of SELECTTAILCALL is to reduce false positives of FunSeeker, and our experimental results show that SELECTTAILCALL indeed helps increase precision by 73.18%.

E. Implementation

To parse ELF exception handling information and disassemble binaries, we used B2R2 [23], which has an efficient and precise binary analysis frontend [24]. In total, we wrote 377 source lines of F# code to implement FunSeeker. We make our tool as well as our datasets public to support open science: https://github.com/B2R2-org/FunSeeker.

V. Evaluation

In this section, we evaluate our tool, FunSeeker, to answer the following research questions.

RQ1. How effective are the methods FunSeeker employed in terms of identifying functions? ($\S$V-B)

RQ2. How does FunSeeker compare to current identification tools in terms of correctness? ($\S$V-C)

RQ3. How does FunSeeker compare to current identification tools in terms of speed? ($\S$V-D)

A. Experimental Setup

1) Obtaining the Ground Truth: We obtain the ground truth about function entry addresses by referring to the DWARF symbols. However, there are some corner cases. During compilation, GCC sometimes extracts several code blocks from a function and makes them a separate function: those newly generated functions are often located far from the original function and have a name with a suffix, such as .cold or .part. Although compilers put a function symbol to them, they are not a function per se, but a “part” of a function in a strict sense. Thus, we exclude them from our ground truth. Additionally, we observed that compilers sometimes miss out a function symbol for the compiler intrinsic function __x86.get_pc_thunk. This happens specifically when the function is called by the _start function. Thus, we manually included it in our ground truth.

2) Comparison Targets: We chose three state-of-the-art tools for comparison: IDA Pro [17] (v7.6), Ghidra [30] (v10.0.4), and FETCH [33] (commit efe138). IDA Pro is a well-known commercial off-the-shelf reverse engineering tool, which employs proprietary heuristics as well as FLIRT [16], a signature-based function identification approach. Ghidra is the most actively developed open-source binary analysis platform. Ghidra aggressively utilizes .eh_frame information [26], [32] to recognize function entries. FETCH is a state-of-the-art function identification tool that leverages exception handling information to detect functions. We wrote a script for IDA Pro and Ghidra to extract functions they found. While running our experiments, we found that Ghidra and FETCH sometimes get stuck in an infinite loop or crash due to out of memory when analyzing some binaries in our dataset. Therefore, a total of 115 binaries were excluded from our dataset.

3) Running Environments: We ran our experiments on an Intel Core i9-11900K processor. To make fair comparisons, we set up a VM with VMWare Pro 15.5.7, and allocated a single core and 8 GB of RAM to each VM. We used Windows 10 for IDA Pro, and Ubuntu 20.04 for the others.

B. Effectiveness of FunSeeker

To understand the effectiveness of FILTERENDBR and SELECTTAILCALL, we measured how the precision and recall scores change with or without them. Specifically, we ran FunSeeker under the following four different configurations.

1) We turn off both FILTERENDBR and SELECTTAILCALL, and simply use the end-branch instructions (E) and direct call targets (C) found by DISASSEMBLE (\(E \cup C\)).

2) We use the same configuration as in 1), but turn on FILTERENDBR to reduce false positives (\(E' \cup C\)).

3) We use the same configuration as in 2), but consider jump targets (J) to be more inclusive (\(E' \cup C \cup J\)).

4) We use the same configuration as in 3), but turn on SELECTTAILCALL to reduce false negatives (\(E' \cup C \cup J'\)).

Table II describes the precision and recall scores of FunSeeker with the four different configurations. The first configuration achieved an over 99% recall rate, while the precision was relatively low. Particularly, SPEC includes C++ binaries, which contain a significant amount of catch blocks starting with an end-branch instruction (as discussed in §III-B). Therefore, FunSeeker misidentified those catch blocks as a function entry with 1.

By turning on the FILTERENDBR module with the second configuration, FunSeeker achieved a precision rate of over 99%. This is because FILTERENDBR can disregard end-branch instructions located at exception catch blocks and indirect-return function call sites. 2) does not change the recall rates, but only improves the precision rates. Thus, we conclude that FILTERENDBR can reduce the false positive rate of FunSeeker without affecting the false negative rate.

With the third configuration, we additionally consider every direct jump target J as a valid function entry. While we can get the highest recall rates with 3, FunSeeker
TABLE III: Function identification results compared to the state-of-the-art tools.

<table>
<thead>
<tr>
<th></th>
<th>FunSeeker</th>
<th>IDA Pro</th>
<th>Ghidra</th>
<th>FETCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binutils</td>
<td>99.482</td>
<td>99.775</td>
<td>91.099</td>
<td>91.213</td>
</tr>
<tr>
<td>Coreutils</td>
<td>99.690</td>
<td>99.268</td>
<td>90.004</td>
<td>70.136</td>
</tr>
<tr>
<td>SPEC 2017</td>
<td>99.358</td>
<td>99.911</td>
<td>89.188</td>
<td>96.372</td>
</tr>
<tr>
<td>Binutils</td>
<td>99.462</td>
<td>99.666</td>
<td>95.364</td>
<td>98.970</td>
</tr>
<tr>
<td>Coreutils</td>
<td>99.671</td>
<td>99.237</td>
<td>97.956</td>
<td>93.652</td>
</tr>
<tr>
<td>SPEC 2017</td>
<td>99.379</td>
<td>99.897</td>
<td>93.885</td>
<td>97.967</td>
</tr>
<tr>
<td>Total</td>
<td>99.407</td>
<td>99.828</td>
<td>95.754</td>
<td>99.194</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Prec. (%)</th>
<th>Rec. (%)</th>
<th>Time (s)</th>
<th>Prec. (%)</th>
<th>Rec. (%)</th>
<th>Prec. (%)</th>
<th>Rec. (%)</th>
<th>Prec. (%)</th>
<th>Rec. (%)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coreutils</td>
<td>99.671</td>
<td>99.237</td>
<td>0.273</td>
<td>97.956</td>
<td>64.409</td>
<td>93.652</td>
<td>98.705</td>
<td>99.633</td>
<td>99.224</td>
<td>0.283</td>
</tr>
</tbody>
</table>

Finally, we turn on SELECTTAILCALL in the fourth configuration to reduce the false positives by identifying tail calls as a function entry. FunSeeker significantly outperforms all the state-of-the-art tools in terms of both precision and recall. We further analyzed cases where FunSeeker failed to correctly identify functions. Out of 15,935 false negative cases, 93.3% of them were a dead function, i.e., they were never used in the program. The rest cases (6.7%) were a tail call target that our SELECTTAILCALL misidentified. We also analyzed the 55,045 false positives cases, and found that all of them were referring to a .part block. Specifically, 57.1% of them were a misidentified tail call, and 42.9% of them had a direct call as if they were a function.

C. Correctness of FunSeeker

We compared FunSeeker against the state-of-the-art tools we chose in §V-A2 in terms of both precision and recall rates. The prec. and the rec. columns in Table III respectively represent precision and recall for each tool. Note FunSeeker significantly outperforms all the state-of-the-art tools in terms of both precision and recall.

We further analyzed cases where FunSeeker failed to correctly identify functions. Out of 15,935 false negative cases, 93.3% of them were a dead function, i.e., they were never used in the program. The rest cases (6.7%) were a tail call target that our SELECTTAILCALL misidentified. We also analyzed the 55,045 false positives cases, and found that all of them were referring to a .part block. Specifically, 57.1% of them were a misidentified tail call, and 42.9% of them had a direct call as if they were a function.

IDA Pro achieves the lowest recall rate. By further analyzing the results, we found that 96% of the false negative cases were due to the failure in identifying indirect branch targets.

Ghidra discovers more function entries than IDA Pro, but achieves the lower recall rate for x86 binaries especially when they do not have a relevant Frame Description Entry (FDE). This implies that Ghidra largely relies on the .eh_frame conventions, to precisely identify tail call targets. This result confirms that FunSeeker is substantially faster than the state-of-the-art tools while achieving the highest precision and recall rates.

VI. LIMITATION AND FUTURE WORK

By design, FunSeeker operates only on CET-enabled binaries. That is, it does not handle legacy binaries. However, we note that CET is enabled by default on modern compilers and OSES. Therefore, FunSeeker will eventually benefit function identification and binary analysis.

Although linear sweep disassembly can achieve nearly 100% instruction coverage for regular binaries as noted by [3], it is not always the case when the binary code contains hand-written assembly or inlined data, which can cause a false positive for FunSeeker. Incorporating recursive disassembly or superset disassembly [7], [29] with FunSeeker to improve instruction coverage is promising future work.

GCC and Clang provide the -mmanual-endbr option to disable automatic end-branch insertion. Instead, they allow the users to manually control where to insert an end-branch instruction through a function attribute. Although this option can affect the precision of FunSeeker, the impact will be marginal. First, all the indirect branch targets should still have an end-branch instruction, because otherwise, the program will crash. Second, since FunSeeker performs a linear-sweep...
disassembly to detect direct call targets, it will still be able to
discover most of the regular functions as discussed in
§V-B. Finally, FunSeeker can only miss some direct tail
call targets and unreachable functions, but their portion is only
about 1.24% according to our study (Figure 3).

Although our main focus was on Intel CET, we believe our
algorithm can be easily extended to handle ARM BTI
instructions because end-branch instructions in both architec-
tures behave almost the same. It is indeed promising future
work to handle BTI-enabled ARM binaries.

VII. RELATED WORK

A. Hardware-assisted Defenses

Memory corruption bugs have been a significant threat to
computer security. Various defense mechanisms have been
proposed to date, but only a few of them are used in practice
due to their performance overhead. Control-Flow Integrity
(CFI) [1] is a representative defense technique that can effec-
tively mitigate control-flow hijack exploits, such as [8], [9],
[31], [41], [44].

Recently, modern CPUs are shipped with security features
to address the performance challenge. Pointer Authentication
(PA) of ARMv8-A [40] detects illicit modification of pointers.
PA generates a cryptographic message authentication code,
named Pointer Authentication Code (PAC), and embeds it
in the unused bits of the pointer. PAC is verified before dereferencing the pointer to ensure its validity. Also, Branch
Target Identification (BTI) is a new instruction introduced by

Memory Protection eXtension (MPX) of Intel provides
hardware-assisted bound checking [21]. MPX includes a set of
new instructions to create, propagate, store, and check pointer
bounds. In addition, Intel’s Memory Protection Key (MPK)
allows a user process to manage its own page table permission.
MPK enables a user to set up a non-readable code page, which
is called an execute-only code page. Control-flow Enforcement
Technology (CET) [22] is the most recent security feature
introduced by Intel. CET provides hardware-level CFI with
minimal performance overhead.

B. Function Identification

Function identification is the cornerstone of binary analysis
and reverse engineering because Control-Flow Graph (CFG)
recovery techniques often rely on the assumption that function
entries are known [43]. For this reason, mainstream binary
analysis tools [17], [30] often employ a set of heuristics to
identify functions. They often combine call graph traversal
with compiler-specific pattern matching to identify function
entries. However, pattern matching in general is not robust
against varying binary patterns.

Machine Learning (ML) based approaches [6], [35], [42],
[46], [49] have been proposed to address this challenge.
ByteWeight [6] builds a prefix tree model to compute the
probability of function start. Shin et al. [42] utilize a bidi-
rectional Recurrent Neural Network (RNN) model to detect
function boundaries. FID [46] extracts semantic features from
each basic block with symbolic execution, and leverages three
machine-learning algorithms to identify functions. XDA [35]
employs a deep learning-based language model to identify
functions. Recently, DeepDi [49] models different relations
between instructions and utilizes such relations to generate a
feature vector. As Koo et al. [26] recently reported, those ML-
based approaches are prone to errors when handling unseen
binary patterns as they are largely dependent on the training
dataset. On the other hand, FunSeeker does not require a
training phase.

Static-analysis-based approaches [4], [39] have also been
proposed to overcome the limitations of the pattern-based
approaches. Qiao et al. [39] examines the statically observable
properties of candidate functions to filter out spurious func-
tions. Nucleus [4] presents a compiler-agnostic function detec-
tion algorithm to find entry points through an intra-procedural
control flow analysis. All these approaches are orthogonal
to FunSeeker, and can benefit from FunSeeker. For
example, one could employ FunSeeker as a preprocessing
step for function identification.

Recently, researchers [2], [33], [37], [38], [48] have started
to pay specific attention to the .eh_frame section to identify
function entries. The .eh_frame section provides a way to
unwind the stack when an exception is raised. This is useful
because modern compilers use frame pointers, e.g., RBP, as
a general purpose register [19], [20]. FETCH [33] systematically
analyzes the .eh_frame section information to be able
to precisely identify functions. Interestingly, however, about
56.3% of the functions in our Clang binary dataset have no
corresponding call frame information (in the .eh_frame
section) and about 3.3% of the FDEs are related to .part or
cold functions, which are not a real function. We believe both
FETCH and FunSeeker are complementary to each other.

VIII. CONCLUSION

In this paper, we demonstrated FunSeeker, a novel func-
tion identification tool that works on CET-enabled binaries.
To build our system, we first systematically analyzed how
Intel CET’s end-branch instructions are used in real-world
binaries. We then extracted several usage patterns to design an
efficient function identification algorithm, whose complexity
is linear in the size of the target binary. Even if the sim-
plistic design, FunSeeker was able to achieve significantly
higher performance compared to the existing state-of-the-art
tools. Furthermore, FunSeeker is highly compatible and
lightweight so that it can be easily adopted by existing tools.
Consequentially, our study confirms that CET can eventually
benefit binary analysis tools.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their feedback.
We also thank Erik van der Kouwe for shepherding our paper.
This work was supported by Institute of Information
& communications Technology Planning & Evaluation (IITP)
grant funded by the Korea government (MSIT) (No.2021-0-
01332, Developing Next-Generation Binary Decompiler).