Now You See Me: Real-time Dynamic Function Call Detection

Franck de Goër
UGA, VU Amsterdam
ANSSI
franck.de-goer@ssi.gouv.fr

Sanjay Rawat
Vrije Universiteit Amsterdam

Dennis Andriesse
Vrije Universiteit Amsterdam
da.andriesse@few.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Roland Groz
LIG
Univ. Grenoble Alpes
roland.groz@univ-grenoble-alpes.fr

ABSTRACT

Efficient detection and instrumentation of function calls is fundamental for a variety of dynamic analysis techniques, including dynamic callgraph construction, control-flow integrity, and automatic vulnerability discovery. A common way of detecting calls at the machine code level is to look for `CALL` instructions. However, optimizing compilers frequently implement function *tail calls* with `JMP` instructions instead, and distinguishing an intra-procedural jump from a `JMP`-based function call is not straightforward. Despite the importance of making this distinction, prior research has not produced a reliable solution. In this paper, we address the problem of dynamic *function call detection* in real-time. We propose a heuristic-based approach named `iCi` to efficiently and automatically instrument calls, including conventional `CALL`s and `JMP`-based calls, at runtime. `iCi` does not rely on source code, debug information, symbol tables or static analysis. We show that `iCi` achieves an f-score of 0.95 in the worst case, regardless of optimization level. We open-source our implementation as well as the oracle we used for our evaluation.[1]

KEYWORDS

Reverse-engineering, dynamic instrumentation, binary analysis

ACM Reference Format:
Franck de Goër, Sanjay Rawat, Dennis Andriesse, Herbert Bos, and Roland Groz. 2018. Now You See Me: Real-time Dynamic Function Call Detection. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3274694.3274712

1 INTRODUCTION

Dynamic binary instrumentation is widely used for reverse-engineering [6], profiling [13], runtime checks and debugging [9]. Many dynamic analysis applications, including security solutions like `binCFI` [19] and `TypeArmor` [16], depend heavily on reliable function call recognition. Unfortunately, conventional function call detection methods which consider only `CALL` instructions are error-prone when used for instrumenting callsites, leading to broken binaries or diminished security. Specifically, these methods miss optimized *tail calls*, which implement function calls using the `JMP` instruction instead of `CALL`. This is common in optimized binaries, such as those compiled with `gcc` at `-O2` or `-O3`. In `coreutils` compiled with `gcc` at `-O2`, the proportion of `JMP`-based calls encountered in the execution is about 10% of all function calls. These include both direct and indirect jumps, and also conditional jumps. Clearly, instrumenting only `CALL` is not enough to efficiently catch all function calls.

Despite the importance of accurate call detection for many dynamic analyses, no prior work solves the issue of dynamic real-time function call detection in dependable and reliable manner. While several works discuss tail call optimization from a compiler perspective [10, 14, 15], none consider the binary analysis point of view. Existing dynamic analysis frameworks, such as `Pin` [8], `Dyninst` [5] and `DynamoRIO` [4] provide only instruction-level syntax-centric APIs, which force the developer to manually instrument all instruction classes of interest. They provide no high-level, semantics-oriented way of simply instrumenting all function calls regardless of the low-level call implementation. As a result, these frameworks are error-prone. For example, `perftools`, based on Intel PT,[2] allows tracing the program execution and further provides the option to filter the trace based on the branches that correspond to function calls only. However, on `ffmpeg` (compiled with `gcc` at `-O2`) it detects only 1109142 calls out of a total of 1467291 function calls.

In this paper, we address the *function call* detection problem using a dynamic heuristic-based approach named `iCi` (Intuitive Call Instrumentation), which can efficiently distinguish `JMP`-based calls from intra-procedural jumps in real-time during execution of x86-64 binaries in the dynamic analysis framework `Pin`. `iCi` does not require any prior knowledge on the binary under analysis, and does not rely on the source code, debug information, symbol table or any static analysis. To evaluate `iCi` and competing solutions, we also develop an open-source oracle. For the *function identification* problem (finding instruction ranges for each function), the oracle is straightforward, as function boundaries are included in the debug information. In the case of the runtime *function call*

[1] https://github.com/Frky/iCi

[2] https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/intel-pt.txt

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274712

detection problem (detecting all function calls, both CALL and JMP-based), ground-truth is not statically available. Section 4.1 discusses how the oracle handles this problem. We evaluate our approach on coreutils, binutils, ffmpeg and evince. For each program, we experiment with four levels of optimizations with gcc. We compare the accuracy and overhead of iCi with two naive approaches one could use for call detection, using our oracle as the ground-truth provider. We also test our function detection on SPEC CPU2006 compiled with -O2. The results show that our approach catches function calls at runtime with an f-score of 0.95 in the worst case (400.perlbench from SPEC CPU2006 compiled with gcc -O2), and does not suffer from optimizations such as tail calls. In comparison, instrumenting only CALL instructions gives an f-score of 0.906 on the same binary. Our results also show that iCi applies to both procedure-oriented and object-oriented programs.

Contributions. First, we define and address the problem of dynamic *function call detection* at runtime. Second, we propose an oracle to obtain ground-truth which can be used for evaluating solutions to this problem. Third, we introduce iCi, a heuristic-based dynamic function call detection approach. Finally, we provide both the oracle and iCi as open-source.[3]

2 PROBLEM

In this section, we first present a concrete example of the problem we address, taken from a real-world application (ffmpeg). Then we introduce notations and definitions to precisely describe the problem. Finally, we define the scope of our work, including the assumptions we make, and those we do not require.

2.1 Statement

The problem we address in this paper is the following: *how to catch, dynamically and in real-time, every call to any function embedded in a given binary?* Or, in other words, from a practical point of view, *what dynamic instrumentation is needed to efficiently achieve this?*

Let us consider an assembly code snippet, shown in Listing 1, from ffmpeg compiled with gcc 5.4 at -O2 optimization level. Although function calls in unoptimized binaries are mainly implemented through the CALL instruction, this changes when optimizations are enabled. For example, the common tail-call optimization emits JMP-based function calls.[4]

```
0000000000460c3f <ff_ac3_float_mdct_end>:
  460c3f:  push   %rbx
  460c40:  mov    %rdi,%rbx
  460c43:  lea    0x570(%rdi),%rdi
  460c4a:  callq  431ee7 <ff_mdct_end>
  460c4f:  lea    0x5e0(%rbx),%rdi
  460c56:  pop    %rbx
  460c57:  jmpq   e2bd50 <av_freep>
```

Listing 1: Example of JMP-based call in ffmpeg

[3] https://github.com/Frky/iCi

[4] Note that Listing 1 is the exhaustive code of ff_ac3_float_mdct_end. The next instruction in the binary is the first instruction of the next function (namely ff_ac3_float_mdct_init).

In this example, we see that the function av_freep is called through a jmpq instruction (at address 0x460c57). This is a concrete example of a tail-call which would not be caught by an instrumentation based on the CALL instruction only. Our experiments, presented in Section 5, show that, for the -O2 and -O3 optimization levels, naive instrumentations of calls either miss a significant number of calls (if we only instrument CALL instructions) or produce a huge number of false positives (if we consider every jump as a call). In particular, for ffmpeg compiled with -O2 by gcc 5.4, the instrumentation of CALL instructions only leads to an f-score of 0.87 and more than 20% of calls are missed.

2.2 Notations and definitions

Now that we introduced briefly, with a concrete example, the problem we target, we propose in this section notations and definitions that lead to a more formal form of this problem and objectives.

2.2.1 The problem. Let us denote by:

- B a binary program that we want to analyze,
- $F(B)$ the set of functions implemented in B or dynamically linked at runtime,
- e a given execution of B,
- $I(e)$ the sequence of instructions executed during e.[5]

From these notations, the problem we address in this paper can be formalized as follows: *for a given execution e of B, and for each instruction i of $I(e)$, determine in real-time if i is an invocation of a function $f \in F(B)$.*

2.2.2 Oracle. We assume that we can construct a post-mortem oracle for each function $f \in F(B)$, that we denote by $O(f, e)$. For a given execution e, $O(f, e) \subset I(e)$ is the list of instructions in $I(e)$ that are invocations of f. By post-mortem, we mean that this oracle does not have to be available in real-time. It can be produced at the end of an execution e.

2.2.3 Inference. For a given execution e and a function $f \in F(B)$, let us denote by $C(f, e) \subset I(e)$ the list of instructions inferred as being invocations of f in real-time. The goal is to find an efficient way to construct $C(f, e)$, with minimal assumptions, and without relying on the source code, the symbol table or debugging information, such that $C(f, e)$ produces the best possible f-score with respect to the ground-truth $O(f, e)$ for each function f.

2.2.4 F-score. To compute the f-score, we need to introduce two additional definitions:

- a *false negative* is an instruction $i \in O(f, e)$ such that $i \notin C(f, e)$,
- a *false positive* is an instruction $i \in C(f, e)$ such that $i \notin O(f, e)$.

For a given execution e, we denote by FN (resp. FP) the set of false negatives (resp. false positives) for any function. True positives are defined as the intersection of $O(f, e)$ and $C(f, e)$ for any function f. From these, *precision* and *recall* are defined as follows: $p = |TP|/(|TP| + |FP|)$ and $r = |TP|/(|TP| + |FN|)$. Then,

[5] Note that each instruction of $I(e)$ is context-dependent; this means that if a static instruction i is executed several times during e, it will correspond to several elements of $I(e)$.

for a given execution e, the f-score of the inference C is given by $2 * p * r/(p + r)$.

2.3 Scope

Let us now define the scope of this paper, and in particular the binaries we target and the assumptions we make.

2.3.1 Binaries. In this work, we consider compiled x86-64 binaries obtained from source code, written either in procedure- or object-oriented languages. We do not make any assumptions on the language of the source code, and our approach generalizes to other architectures. In addition, the approach does not rely on any debug information, nor on the symbol table or the string table. In other words, *the scope of this paper is any binary program obtained by compilation, and does not require recompilation.* Another important point is that we do not rely on any static analysis of the binary. Our approach can be implemented even if we only have access to a stream of instructions at the exact moment they are executed (i.e., in real-time).

2.3.2 Assumptions. Although we do not rely on assumptions regarding information that we can get from the binary to analyze, we do make several assumptions on the compiler's choices.

- **single entry points** - we assume that functions have a unique entry point in the compiled binary. Our experiments and recent work [1, 2] show no evidence of counter-examples with `gcc` or `clang`, although there may be examples in hand-crafted code such as in `glibc`.
- **no interleaving** - we assume that functions are not interlaced in memory. This means that the sets of instructions delimited by the entry point and the last return point of each function should not overlap. Again, we found no counter-examples in our compiler-generated dataset.
- **return address on the stack** - we assume that the return address is at the top of the stack just after a `CALL` instruction occurs.
- **non-obfuscated code** - in this paper, obfuscated binaries are out of scope.

Note that even if some assumptions do not always hold, counter-examples have a low impact on our results as long as they are the exception and not the rule. Moreover, while counter-examples may lead some heuristic-based checks to fail, the knowledge of `iCi` improves as the binary executes, allowing it to eventually catch and correct cases where assumptions are violated (see Section 3).

3 APPROACH

In this section, we propose an approach to answer the problem we defined in the previous section. In particular, we provide heuristics to decide, for a given `JMP` instruction, if it should be considered as a call or not. In addition, we want our approach to be *real-time*. This means that, for each jump, we want to decide *at the moment it is executed* how to classify it. This requirement is due to the fact that we want this approach to be applicable for real-time analysis such as CFI.

As we do not want to rely on the symbol table, we do not assume to know function boundaries at the beginning of the execution. In addition, we recall that we do not assume to be able to analyze statically the binary before the execution at this point. Therefore, the analysis is only based on the flow of instructions that are actually executed by the CPU.

This approach is implemented in a tool named `iCi`, whose implementation details are given in Section 4.3.

3.1 Overview

In a few words, our approach consists of two main parts: catching obvious calls, and filtering jumps to decide which ones are calls and which are not. Each of these parts could lead to the detection of a call. Each time a call is detected, we add the target to the list of known functions. This list is thus enhanced over the execution. Although this is not the purpose of this paper, the results can be used to detect functions as well.

3.1.1 Catching obvious calls. We consider two categories of instructions to be calls with no further investigation. The first one is the `CALL` instruction, either with a direct or an indirect target (because we are dynamic, we can always resolve the concrete address of the target). The second one is any `JMP` instruction located in the `.plt` section of the binary: *every jump of this kind is immediately considered to be a call* and thus *does not go through our filter process* presented in the next section. Knowing that a given instruction is located in the `.plt` section requires knowledge of the section layout. Let us make two remarks regarding this assumption. First, the section names are not removed when a binary is stripped, so it is safe to assume their presence within the scope of our work as outlined in Section 2.3.1.[6] Second, while our special treatment of jumps from the `.plt` is needed to comply with our oracle's strict ground truth definition (see Section 4.1), many applications in practice do not suffer much if `JMP` instructions from the `.plt` are missed (*i.e.*, not considered to be calls). Indeed, the `JMP` instruction is just a wrapper from a `CALL` instruction to the actual code to be executed in a library function. Missing it does not mean missing the call to the library function.

3.1.2 Filtering jumps. The second part of our approach decides in real-time, for any `JMP`-based instruction[7] (*i.e.*, both unconditional and conditional jumps, and both direct and indirect jumps), whether it is a call or not. Figure 1 illustrates the different steps we perform on each `JMP` instruction for this purpose. We propose several heuristic-based checks that can be split into two categories: **exclusion checks** - if one of these checks fails, then the `JMP` is *not* considered to be a call; and **inclusion checks** - if one of these checks passes, then the `JMP` *is* considered to be a call. We perform inclusion checks *after* exclusion checks. In addition, if neither of these two categories of checks lead to a conclusion, then we apply a default policy which can be configured in our implementation. In our experiments, we use a default policy which considers a `JMP` instruction as *not a call*.

Another important point is that, during the execution, we keep a memory of the previous decisions. Every time a call is detected (either based on the `CALL` instruction, a jump from the `.plt` or a `JMP` instruction that passes an inclusion check), the target of this call

[6] Even if section names are not available, it is straightforward to find out which section is the `.plt`, since it has a very particular structure.
[7] Except jumps from the `.plt` section

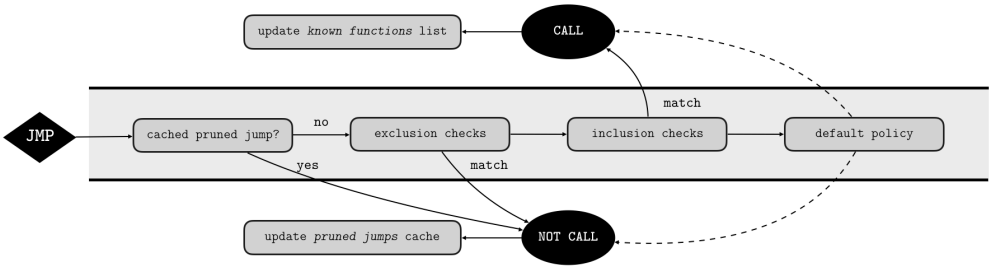


Figure 1: Overview of the different steps to conclude on the nature of a `JMP` instruction

is added to a list of *known entry points*. This is important, because it means that the further the execution proceeds, the more entry points we know, and the more accurate and efficient our detection becomes. The same goes for negative checks: if a `JMP` instruction matches an exclusion check, then we add it in a structure that caches the pruned jumps. This allows us to decide quickly on jumps that we have already seen. The implementation details on this particular point (jump caching) are given in Section 4.3.4.

In the next section, we present our heuristics used in every check we perform.

3.2 Heuristics

We keep track of the following data, accessible at every jump instrumentation:

- the current program counter `%rip`,
- the target of the jump (address),
- the current entry point, which is the target of the last instruction that was detected as a call (either `CALL`, `JMP` from `.plt` or any `JMP` instruction that matched an inclusion check). Note that this data might be inaccurate at some point, for instance if we missed the last `JMP`-based call,
- the current state of the stack (*i.e.*, value of `%rsp`),
- the state of the stack when the last call occurred,
- information about the return point of functions seen so far (see Section 4.3.3).

The first two items of this list are straightforward to know, as we are performing dynamic instrumentation. The current entry point depends on the accuracy of the previous instrumentation, as discussed. We give details about the stack information in Section 4.3.2, but for now consider that this information is available but not completely accurate.

3.2.1 Exclusion checks. We have four exclusion checks, that are performed sequentially. If one of them fails, then the jump is considered as *not a call* immediately and further checks are not performed.

* **Jumps from `.plt` to `.plt`** – this is to avoid internal jumps in `.plt`. These internal jumps are related to the dynamic loading process, and we are not interested in catching them. This check only relies on the knowledge of the `.plt` location, as discussed before.

* **Internal jump before** – if the target of the jump is between the current entry point (*i.e.*, the target of the last detected call) and the current `%rip`, then it is an internal jump. This relies on two of the assumptions we presented in Section 2.3.2: functions have a single entry point and they are not interlaced in memory.
Discussion: This check relies on both assumptions and previously inferred data. Wrong assumptions would be bad, as they would cause `JMP`-based calls to be pruned whereas they should not be. For example, if a function has several entry points, then a recursive `JMP`-based call to the second entry point will not be seen as a call but as an internal jump. On the other hand, inaccurate data could also produce false negatives. For instance, if the current entry point is wrong and points to a lower address, we would prune a `JMP`-based call to a function located between the jump instruction and the wrong entry point. Listing 2 illustrates this situation (this example was taken from `ffmpeg`). However, our experiments show that this is contained in a very few number of false negatives in practice.

```
000000000040ed24 <uninit>:
  /* supposed entry point */
...
000000000040ed47 <init>:
  /* target of the jump */
...
000000000040ee41 <init_alphaextract>:
  /* real entry point */
  40ee41:  mov    0x48(%rdi),%rax
  40ee45:  movl   $0x8,0x8(%rax)
  40ee4c:  jmpq   40ed47 <init>
```

Listing 2: Example of a wrong entry point leading to a false negative.

* **Internal jump after** – if the target of the jump is between the current `%rip` and a known return point of the current function, then it is an internal jump. Section 4.3.3 presents our implementation internals to get an approximation of these return points for each function.
Discussion: The previous discussion about inaccurate data also applies here. If we misdetect the function in which the execution currently is, then the corresponding return point can be wrong. If so, this check can produce false negatives as well. The method we use to retrieve return points (see Section 4.3.3) can also lead to inaccurate values. It is important to note, however, that we initialize the return point of a new function to the same value as the entry point. This implies that this check has no effect on functions for which the return point is not discovered yet.

* **Stack inconsistency** – check if the state is the same as it was when the last call occurred. Indeed, for a *tail call* to happen correctly,

the stack must be cleaned first and the top of it should contain the (same) return address that was pushed before the previous call.

Discussion: Here, we implicitly assume that JMP-based calls are in fact *tail calls*. This assumption is compliant with both our experiments, and prior work [1]. This check is fundamental: every further inclusion check will be performed on jumps assuming that the state of the stack is consistent with a call. This allows us to prune a lot of jumps with a simple check.

3.2.2 Inclusion checks. Remember that inclusion checks are performed only if no exclusion check was conclusive. Thus, at this point, we know in particular that the stack is consistent with a call, and that we are not in the `.plt`.

* **Known entry point** – this simply checks if the target of the jump is a known entry point of a function. Recall that, whenever a call is detected (either a `CALL` instruction, a jump from the `.plt` or a `JMP`-based instruction inferred as a call in a previous check), its target is added to a list of known functions.

Discussion: Although this check is implemented in an efficient way (see Section 4.3.1), we still perform it after exclusion checks to properly handle a particular case of a jump to the entry point of the current function. This could either be a loop starting at the entry point of a function or a recursive call. To distinguish these cases, we need to first check the stack consistency.[8]

* **External jump before** – this is to check if the target of the call is before the last known entry point. Assuming that functions have a single entry point, this means that we are leaving the current function, therefore it is a call.

Discussion: This is the complementary test to the *internal jump before* exclusion check and therefore, it also relies on the same assumptions and data values that could be inaccurate.

* **Cross entry point** – this checks if there is any known entry point between the current `%rip` and the target of the jump. If so, then we consider this jump as a call. It is the most costly check, even though we optimized the implementation to reduce this cost as much as possible (see Section 4.3.1). That is why this check is performed last.

Discussion: This is the most fundamental inclusion check. It allows us to correctly detect the vast majority of calls based on `JMP` instruction. Once again, the more the execution goes, the more entry points we know, and therefore the more likely a `JMP`-based call will cross one of them.

3.2.3 Default policy. As mentioned before, if none of the checks is conclusive, then we apply the default policy. In our implementation, the default policy is to prune the jump. However, in this case, because there could be a misdetection due to inaccurate data, we do not add the pruned jump to the cache. Thus, next time the same jump is encountered, all the checks will be performed again.

4 IMPLEMENTATION

We now present some details about the `iCi` implementation. First, we give details about the way we obtain ground-truth. Second, we describe two naive implementations that we use to evaluate against. Third, we give technical details about the way we implement our approach.

[8] We encountered such cases in `ffmpeg` compiled with `-O2`.

4.1 Ground-truth - `oracle`

In Section 2.2, we presented a formal definition of the ground-truth. In this section, we propose a way to construct an oracle that provides it.

4.1.1 Entry points. The first step to construct our oracle is to get entry points of functions. We use the information from the symbol table to achieve that. In addition, we compute the addresses of entries in the `.plt` section, and consider them to be entry points of functions. Finally, at run time, we instrument routines (using `Pin`) to get entry points of functions from dynamically loaded libraries.

4.1.2 Call detection. The oracle instruments every instruction, and for each instruction checks if the program counter (`%eip`) is sequential or not. If a discontinuity is detected, the oracle checks if the new value of the program counter corresponds to the entry point of a function, and if so it increments the number of calls.

4.1.3 Discussion. The oracle considers each hit of the entry point of a function as a call, if it follows a control-flow discontinuity.[9] This merits discussion in two scenarios: if a function loops to its own entry point, or if a function is called without discontinuity of the program counter. The first case will be seen as a call from the oracle perspective, while the latter case would not. In practice, these cases do not happen often with compiled code: function prologues avoid the first case, while the second is statistically very unlikely.

Additionally, our oracle is not efficient and causes high overhead. This is indirectly intended: we did not want to add any more complexity than necessary in the oracle, to be sure we are as close as possible to the oracle definition from Section 2.2.2. For example, every instruction is instrumented, and the only check is regarding a discontinuity of the program counter. Moreover, the high overhead is not an issue, as the oracle is intended for evaluation purposes only.

4.1.4 Comparison with the oracle. From the oracle, we get the total number of calls in one execution that we should detect for every function. Every other analysis we propose in this paper compares to this one. A call detected by the oracle and missed by a given analysis is a *false negative*, and a call detected by the analysis that is not present in the oracle is a *false positive*.

4.2 Naive implementations of call detection

In addition to the oracle, we provide two implementations that correspond to naive approaches to catch calls. We use them in our experiments (see Section 5) to emphasize the problem we address in this paper. These implementations show that naive approaches are not enough to address the problem properly.

4.2.1 `jcall`. The first naive way of implementing call detection, that we name `jcall`, is based on two instrumentations. First, we instrument every `CALL` instruction. Second, we consider every jump from the `.plt` as a call. The second instrumentation catches every call to dynamically loaded libraries. Table 1 shows that in the majority of our tests, this approach produces no *false positives* (exceptions are discussed in Section 5.6). It also shows that this

[9] *i.e.*, when the program counter is not exactly incremented by the length of the current instruction being executed.

approach gives good results at `-O0` and `-O1`, but produces many false negatives at `-O2` and`-O3`.

4.2.2 `jmp`. The previous implementation produces no *false positives*, but misses calls (and in particular tail calls). The second approach we propose is much more conservative: it considers every `JMP` as a call (in addition to `CALL` instructions). Clearly, this produces a lot of false positives - see Table 1. On the other hand, the `jmp` instrumentation produces no *false negatives*, which means that, in our benchmark, all function calls at the assembly level are implemented by either a `CALL` or a `JMP` instruction (conditional or unconditional).

4.3 Implementation details of `iCi`

This section discusses particular points of the `iCi` implementation.

4.3.1 Function information. Each time a call is detected, we store information about the called function. If the target is not known yet, then we need to add an entry to a data structure, in order to detect later calls easily. This entry also stores the number of times a function is called (field `calls`), plus a linked list of instructions (field `ins`) that caused the function to be called (for debug and diagnostic purposes). In addition, it contains information about known boundaries of functions (especially the highest return site), as discussed in the next section. An entry is described by the `C` structure given in Listing 3.

```
typedef struct fn_entry {
    ADDRINT entry;
    ADDRINT ret;
    UINT64 calls;
    string *name;
    string *img_name;
    ADDRINT offset;
    ins_t *ins;
} fn_entry_t;
```

Listing 3: `C` structure of an entry corresponding to a function

Hash table. To efficiently access the entry corresponding to a given function by its entry point, we store this information in a *hash table*. This hash table is indexed by the twenty least-significant bits of the address after a right shift of 4 bits.[10] From this hash table, we can check quickly if a given address corresponds to an entry point of a known function, and if so access the information related to this function.

Binary search tree. In addition, for the last inclusion check (*cross entry point*), we need a way to determine if there is a known entry point between two addresses. To do so, the hash table is not efficient, so we maintain, in addition, a *binary search tree* that stores every known function entry point. This data structure allows us to efficiently check if there is a known entry point between the target of a jump and the address of the jump instruction.

4.3.2 Call stack. For several purposes, and in particular the *stack inconsistency* check, we need to keep a call stack. For this purpose, every time a call is detected, we push a new entry on the top of our internal *stack* structure. This entry stores the following information:

- the target of the call (*i.e.*, the entry point of the function being called),
- the supposed return address (*i.e.*, the address of the instruction statically following the instruction causing the call),
- the current value of `%esp`.

In addition, we instrument every `RET` instruction. When such instructions are executed, we unstack entries from our internal call stack, until one of them has a return address that corresponds to the target of the `RET` instruction. Every function that is unstacked is considered to be returning as well, which allows us to detect function boundaries (see next section). If, for example, a tail call is missed, the stack of calls is not updated and thus gives wrong information about the context. However, this is corrected when the tailcalled function returns.

4.3.3 Return points. One of the checks (*internal jump after*) relies on knowledge of the return boundary of the current function. As mentioned in the previous section, we instrument `RET` instructions to keep an internal call stack. Each time a function returns with the instruction `RET`, we update its return site information. We know which function is returning by looking at the top of the call stack. Although this information might be inaccurate, experimental results show that it is sufficient in practice (see Section 5).

There are two important points to be discussed. First, while it may seem like we make the assumption that functions have a single return site, we do not. We only keep the value of the highest address corresponding to a return site for each function. This makes sense because we assume that functions are not interlaced. Second, we initialize the return site address to the address of the entry point. This ensures that the *internal jump after* check will not produce false negatives on functions for which we do not have a sufficient knowledge yet.

4.3.4 Caching. We also maintain a cache of the jump instructions that were pruned before in the execution. To do so, we use another *hash table*. Each time an exclusion check matches, the pruned jump address is added to the hash table. Therefore, for every jump instruction, before doing any exclusion check, we look for the jump in the hash table, and if it is present we prune it once again without performing any further tests. Note that we do not cache jumps that were pruned by the default policy. For those, all checks are performed again. The reason for this is that the default policy applies when we have no evidence that allows us to reach a conclusion given the context we know. The next time, one of the checks might be conclusive.

5 EXPERIMENTS

This section presents our experiments, to show that `iCi` is accurate in detecting calls, and presents a reasonable overhead. We also show that the results of `iCi` do not suffer from optimization at compilation, and that our approach is compatible with object-oriented programs. Finally, we provide experiments on different compilers, to illustrate the fact that `iCi` is not compiler-specific.

[10] Because of memory alignment, many functions have an address with least-significant bits set to zero.

5.1 Methodology

5.1.1 Benchmark. The general experiments we present in this paper are conducted on 98 `coreutils` programs[11], 13 `binutils` programs[12], `ffmpeg` and `evince`. For each of these programs, we provide an arbitrarily chosen input. A list of all the inputs we use is included in the repository of the tool, as is everything needed to re-run the experiments we present in this paper. Each program is compiled using `gcc` with each level of optimization (from `-O0` to `-O3`). In addition to the general experiments, we run `iCi` on `SPEC CPU2006`. We present the corresponding results in Section 5.4. We do not include our SPEC results in the general experiments because we only evaluate SPEC at the `-O2` level of optimization (see Section 5.4 for more information). Finally, we perform a comparative evaluation of our results on two compilers: `gcc-6.0` and `clang-3.8`, using the `coreutils` programs (see Section 5.5).

5.1.2 Exclude libraries. For each test, we do not track calls in dynamically-loaded libraries (*i.e.*, calls from a library function to a library function) for two reasons: first, two different programs sharing the same library would have correlated results; second, the libraries have the same optimization level for each experiment, and so including them would influence our comparative results. Note, however, that our tool is able to instrument calls within libraries as well.

5.1.3 Comparison. For every program included in our benchmarks, we perform each of the analyses we presented in this paper: `oracle`, `jcall`, `jmp` and `iCi`. We compare the latter three with the `oracle` for accuracy measurements, and for each we compute the f-score. Note that the comparison with `oracle` is performed instruction-wise. This means that not only do we compare the number of calls each method detects, but we also compare each and every instruction that was detected as a call. A perfect f-score for a given analysis *a* means that *a* detected the exact same call instructions as the oracle did, and for each of them the exact same number of hits.

5.1.4 One execution. We want to compare results of different analyses, while programs may have non-deterministic behavior. To solve this practical problem, our accuracy experiments perform each of the analyses during the same execution. Because they do not interfere, this has no influence on the results, and we are able to compare the results between two approaches from a single execution.

5.1.5 Overhead. To measure overhead, we need a different setting. For that purpose, we run one new execution for each program and for each analysis to perform. This way, we obtain execution times and overhead numbers for each individual analysis. We compute the overhead compared to the `jmp` instrumentation. The rationale in comparing to it is as follows. We showed in Section 2.1 that one needs to consider `JMP`-based instructions to achieve complete function call detection. The question is thus how to efficiently and accurately distinguish `JMP`-based calls from intra-procedural jumps. The `jmp` analysis is a good basis for comparison, as it instruments every jump but does not perform any selection (every jump is considered to be a call). Measuring the overhead compared to `jmp` is thus equivalent to measuring the overhead due to `iCi`'s jump selection.

5.2 Platform

All tests are performed on a 64-bit `Debian Stretch (9)` running Linux kernel version `4.9.51-1`. The machine is equipped with an `Intel Core i7-4610M` CPU and 16GB of RAM. We use `gcc-6.3.0`, and `Pin 3.4`, except for `SPEC CPU2006` for which we use `gcc-5.4.0` for compatibility reasons (see Section 5.4).

5.3 General results

Exhaustive results of our experiments on the four sets of program mentioned previously are given in Table 5. In particular, for the four levels of optimization, we present the absolute number of calls[13] (TP), false positives (FP) and false negatives (FN) due to each approach. Number in parenthesis are relative to instructions, by opposition to the main numbers that are relative to calls. For instance, at `-O0`, `jcall` misses 71 calls in total on `binutils`, and these 71 calls are caused by 37 different instructions (some of them are hit several times during the execution).

The following subsections emphasize interesting results of our experiments, based on the results of Table 5. For more clarity, we include partial tables extracted from the main table.

5.3.1 F-score. Table 1 presents the f-score of the different approaches. First, note that, as expected, `jmp` gets a very low f-score: except for `evince`, it never obtains a better score than `0.518` (that is for `coreutils` compiled with `-O1`). Second, our approach, `iCi`, obtains the best f-score in every scenario we encounter, and is never lower than `0.985`. For `binutils`, the average f-score is a perfect `1.000`. For `ffmpeg`, which includes object-oriented code, the f-score is `0.997`. These results also show that the accuracy of `iCi` does not suffer from optimization at compilation time, whereas other implementations (and especially `jcall`) do. For instance, `jcall` gives good results on `ffmpeg` at `-O0` and `-O1`. However, with `-O2`, its f-score is only `0.874` whereas our approach gives an f-score of `0.997`. In Table 5, we observe that the errors due to `iCi` are mostly due to false positives. Indeed, the number of false negatives is always lower than 10 in absolute numbers. From this, we deduce that improving these results would hinge on improving the exclusion checks presented in Section 3. We also observe that the number of false positives in `iCi` is small compared to the number of call sites that cause these errors. For instance, on `coreutils` at `-O0`, we detect 55897 false positives in total (over all benchmarks), but they are caused by only `108` unique instructions.

5.3.2 Overhead. Figure 2 shows the overhead of each approach compared to `jmp`, as mentioned in Section 5.1.5. In addition, we present measurements of a vanilla execution without `Pin`, named `noinst`, plus an execution with `Pin` but no function detection instrumentation (`pempty` for `pintool-empty`), to separately measure the overhead due to `Pin` and the `iCi` instrumentation on top of `Pin`, respectively. Note that `evince` is excluded from these results,

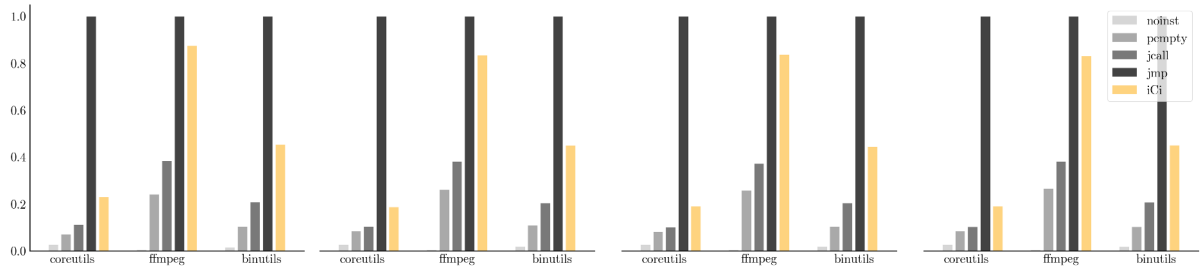
[11] We excluded `runcon`, `chcon`, `nice` and `nohup` because they execute another program with special contexts, `chroot` because it requires `root` privileges, and `yes` for termination purposes

[12] Programs targeting Microsoft Windows have been excluded

[13] According to the oracle

	-O0			-O1			-O2			-O3		
	jcall	jmp	iCi	jcall	jmp	iCi	jcall	jmp	iCi	jcall	jmp	iCi
binutils	1.000	0.223	1.000	1.000	0.222	1.000	0.999	0.222	1.000	1.000	0.221	1.000
evince	0.988	0.831	0.991	0.980	0.796	0.985	0.936	0.803	0.985	0.947	0.818	0.986
coreutils	0.998	0.518	0.999	0.998	0.352	0.998	0.946	0.405	0.998	0.997	0.351	0.998
ffmpeg	0.948	0.227	0.997	0.919	0.183	0.997	0.874	0.182	0.997	0.884	0.181	0.997

Table 1: F-score of our approach on binaries compiled with `gcc`



(a) -O0 **(b) -O1** **(c) -O2** **(d) -O3**

Figure 2: Overhead of each analysis in comparison with `jmp` for `-O2`

because it requires interaction with the user, which has an influence on the execution time.

The results confirm that `jcall` is faster than `jmp`, because it does not instrument jumps (except from `.plt`). For instance, on `ffmpeg` compiled with `-O0`, `jcall` is 60% faster than `jmp`. They also show that `iCi` is always faster than `jmp`, from 10% with `ffmpeg` at `-O0` up to 80% with `coreutils` at `-O1`. Although this result might be surprising, it can be explained by our implementation of `iCi`. As mentioned in Section 4.3.4, we use a cache to prune jumps that we have already seen. This is efficient if jumps are executed several times, in which case the overhead of `iCi` is reduced. However, `jmp` does not do any caching, so the more a given jump instruction is reused, the faster `iCi` is in comparison with `jmp`. Compared to `noinst`, the overhead induced by `iCi` is significant (up to 90× for `ffmpeg` at `-O0`), but if we compare it with the overhead due to `Pin` with no instrumentation (`pempty`), the overhead is between 2× and 4×.

Table 2 presents the standard deviation of the execution time of `iCi` over ten executions. These experiments are conducted on all the programs of `coreutils`: each is executed ten times, and we then compute the standard deviation of the execution time. We present the min, max and average standard deviation for all programs (outliers excluded). For example, the standard deviation of the execution time with `iCi` is `0.011` on average for `coreutils` compiled with `-O0`. This shows that `iCi`'s performance is stable.

5.4 SPEC CPU2006

In addition to the general results we presented in the previous section, we also tested our approach on every program of `SPEC CPU2006` written in `C` or `C++`, which we compiled with `gcc-5.4` for compatibility reasons. These results are not included in the general results because we did not run `SPEC` for the four levels

	-O0	-O1	-O2	-O3
min	0.001	0.001	0.001	0.001
max	0.088	0.107	0.102	0.154
average	0.011	0.009	0.008	0.010

Table 2: Standard deviation of the execution time of `iCi` on `coreutils` programs

of optimization, but only for `-O2`. Level `-O2` is the one that gives the worst results in our general experiments, so it makes sense to test `SPEC` on this level. In addition, `-O2` is the default level of optimization when compiling `SPEC` with default configuration files. Table 3 presents the f-score of the two naive approaches (`jcall` and `jmp`) plus `iCi`. In all tests but one, our approach gives an f-score of 1.000. In comparison, on several programs, `jcall` scores significantly lower. For example, on `445.gobmk`, it has an f-score of `0.883` whereas `iCi` has an f-score of `1.000`. As several programs of `SPEC CPU2006` are written in `C++`, this experiment shows that `iCi` performs well on object-oriented code.

5.5 Influence of the compiler

Table 4 presents the f-score of `iCi` and other analyses on all `coreutils` programs compiled with both `gcc-6.3` and `clang-3.8`. These experiments show that, although `jmp` achieves different results with the two compilers (for instance, at `-O1`, it has an f-score of `0.518` with `gcc` and `0.360` with `clang`), `jcall` and `iCi` have very similar results on each compiler.

5.6 Discussion

`iCi` provides the option to run in a verbose mode for comparison to other approaches. This mode outputs every instruction mismatch

	-O2		
	jcall	jmp	iCi
400.perlbench	0.906	0.260	0.947
401.bzip2	1.000	0.014	1.000
403.gcc	0.960	0.183	1.000
429.mcf	1.000	0.015	1.000
433.milc	1.000	0.201	1.000
444.namd	1.000	0.001	1.000
445.gobmk	0.883	0.004	1.000
447.dealII	1.000	0.298	1.000
450.soplex	0.995	0.049	1.000
453.povray	0.995	0.276	1.000
456.hmmer	1.000	0.127	1.000
458.sjeng	0.967	0.089	1.000
462.libquantum	0.909	0.632	1.000
464.h264ref	1.000	0.181	1.000
470.lbm	1.000	0.044	1.000
471.omnetpp	0.848	0.432	1.000
473.astar	1.000	0.073	1.000
482.sphinx3	0.914	0.644	1.000
483.xalanbmk	0.895	0.205	1.000

Table 3: F-score of `iCi` on `SPEC CPU2006`

between the oracle and a given analysis. This is particularly interesting to understand the source of errors. For instance, one might ask why the `jcall` instrumentation leads to false positives, even though it only instruments `CALL`. Listing 4 presents an extract from the output when comparing the oracle with `jcall` for `evince` at level `-O0`.

```
  ...
  (evince@0x42e492) call 0x41cec0 - 0 time [seen 1]
  (evince@0x4431b7) call 0x41cec0 - 0 time [seen 20]
  (evince@0x440660) call 0x41cec0 - 0 time [seen 1]
  (evince@0x44f89d) call 0x41cec0 - 0 time [seen 1]
*(@00000041cec0)  - 105 extra calls
...
```

Listing 4: Partial output of misdetection from `jcall` with `evince` at `-O0`

Each line of this output corresponds to a given instruction causing a misdetection. The last line is the total number of errors for the given function `@00000041cec0`. In this case, this address does not correspond to any function entry point, but to an instruction in the `.plt.got` section. False positives occur because `jcall` considers every `CALL` instruction to be a function call, even those targeting non-function addresses. Due to the page limit, we cannot include a detailed discussion of all interesting cases we encountered with `iCi`. However, we can summarize that the majority of false positives in `iCi` are due to similar cases, namely `CALL` instructions targeting an address that is not a function entry point.

6 APPLICATIONS

Our methodology for real-time dynamic function call detection has practical applications in several security domains. In this section, we discuss how `iCi` applies to three popular research directions: *Control-Flow Integrity*, *automatic vulnerability discovery*, and high-level *Dynamic Binary Instrumentation*.

Control-Flow Integrity – Control-Flow Integrity (CFI) is a popular technique which aims to prevent control-flow hijacking attacks by instrumenting indirect control transfers, including indirect calls and jumps, and checking that they adhere to the application's intended control flow graph (CFG). Binary-level CFI systems often rely on dynamic binary instrumentation (DBI) to transparently enable defenses [3, 11, 16–18]. Many CFI systems protect call sites by ensuring that they target a legitimate function [11, 16–18]. Some go further, keeping a *context* of all indirect control transfers and using this context to judge the validity of new control transfers [3]. In both cases, accurate call site detection, including detection of `JMP`-based indirect calls, is an important primitive. Particularly, the ability to accurately distinguish intraprocedural indirect jumps from `JMP`-based calls enables CFI systems to correctly determine which type of security constraints to add where. This allows CFI systems to enforce more stringent constraints, forcing intraprocedural indirect jumps, such as switch invocations, to stay within the current procedure, while allowing `JMP`-based calls to target only legitimate functions. As explained in Section 1, existing DBI frameworks like Intel Pin do not provide facilities to distinguish intraprocedural from interprocedural indirect jumps [8]. This forces CFI systems to protect `JMP` instructions with catch-all instrumentation that provides less stringent security. `iCi` provides more accurate call site detection, which can distinguish `JMP`-based calls from other types of jumps, directly benefiting the security of CFI approaches.

Automatic Vulnerability Discovery – Many automatic vulnerability detection systems operate at the function level, both for ease of analysis, and because it is a suitable search-granularity for common bugs, such as stack-based bugs [7, 12, 20]. Operating at the function level is also useful for interoperability with other binary analysis primitives, such as symbolic execution, which are powerful tools for semantic analysis but do not scale to full binaries [7]. `iCi` benefits binary-level vulnerability detectors by providing them with a more accurate record of function invocations and more accurate stack-frame information. This enables more powerful bug-search heuristics and root-cause analysis when a vulnerability is found.

High-level Dynamic Binary Instrumentation – In addition to aiding the accuracy of specific types of DBI-based tools, `iCi` also eases the development of such tools. Existing DBI frameworks like Pin [8], Dyninst [5], and DynamoRIO [4] are all very low-level, shifting the burden of problems like function call detection to the developer. `iCi` provides the beginnings of a higher-level API, that allows developers to simply "instrument all calls" with a single API call, without having to worry about the low-level details of how these calls are implemented (with a `CALL` or `JMP` instruction). This allows for more rapid prototyping of DBI-based tools. We intend to pursue other aspects of high-level DBI in future work.

7 CONCLUSION

Efficient and accurate function call detection is the basis of many dynamic program analysis techniques. We present the first high-level approach, called `iCi`, to automatically instrument both `CALL` and `JMP`-based calls in real-time. Compared to naive solutions, such as instrumenting all `CALL` and `JMP` instructions, `iCi` provides a far

	-O0			-O1			-O2			-O3		
	jcall	jmp	iCi	jcall	jmp	iCi	jcall	jmp	iCi	jcall	jmp	iCi
gcc-6.3	0.998	0.518	0.999	0.998	0.352	0.998	0.946	0.405	0.998	0.997	0.351	0.998
clang-3.8	0.997	0.360	0.998	0.997	0.359	0.998	0.997	0.359	0.998	0.997	0.359	0.998

Table 4: F-score of `iCi` on `coreutils` compiled with both `gcc-6.3` and `clang-3.8`

		-O0			-O1			-O2			-O3		
		jcall	jmp	iCi	jcall	jmp	iCi	jcall	jmp	iCi	jcall	jmp	iCi
binutils	TP		18617150			18336085			18327919			18199970	
	FN	71 (37)	0 (0)	0 (0)	71 (37)	0 (0)	0 (0)	21187 (96)	0 (0)	1 (1)	1923 (75)	0 (0)	1 (1)
	FP	7162 (223)	129827016 (20549)	7162 (223)	7162 (223)	128688861 (20037)	7162 (223)	7161 (222)	128642723 (20036)	7162 (223)	7161 (222)	128308957 (20226)	7162 (223)
	fscore	1.000	0.223	1.000	1.000	0.222	1.000	0.999	0.222	1.000	1.000	0.221	1.000
	ovhd	0.207	1.00	0.454	0.203	1.00	0.448	0.202	1.00	0.444	0.206	1.00	0.450
evince	TP		11515			6889			6915			7577	
	FN	63 (37)	0 (0)	1 (1)	63 (37)	0 (0)	1 (1)	652 (180)	0 (0)	1 (1)	589 (148)	0 (0)	1 (1)
	FP	210 (103)	4698 (2179)	210 (103)	215 (108)	3537 (1938)	215 (108)	198 (98)	3399 (1911)	215 (108)	200 (108)	3376 (2092)	215 (116)
	fscore	0.988	0.831	0.991	0.980	0.796	0.985	0.936	0.803	0.985	0.947	0.818	0.986
	ovhd	n.c.	1.00	n.c.	n.c.	1.00	n.c.	n.c.	1.00	n.c.	n.c.	1.00	n.c.
coreutils	TP		31235860			14185835			16299089			12966483	
	FN	37952 (336)	4 (4)	4 (4)	319 (297)	5 (5)	5 (5)	1621545 (654)	5 (5)	11 (11)	24910 (589)	4 (4)	10 (10)
	FP	55897 (108)	58039825 (18199)	55897 (108)	50490 (100)	52287208 (15932)	50490 (100)	50481 (101)	47892488 (15808)	50490 (107)	50480 (108)	47929614 (16101)	50489 (114)
	fscore	0.998	0.518	0.999	0.998	0.352	0.998	0.946	0.405	0.998	0.997	0.351	0.998
	ovhd	0.110	1.00	0.228	0.103	1.00	0.186	0.099	1.00	0.189	0.102	1.00	0.189
ffmpeg	TP		2331517			1484918			1467291			1421025	
	FN	217619 (29)	27 (23)	0 (0)	217666 (12)	15 (11)	0 (0)	322788 (223)	37 (27)	6 (4)	290067 (180)	36 (23)	6 (4)
	FP	13658 (37)	15834525 (12276)	14954 (42)	6682 (25)	13294818 (11145)	7978 (30)	6680 (24)	13183828 (10935)	7978 (30)	6680 (24)	12831611 (11449)	7978 (30)
	fscore	0.948	0.227	0.997	0.919	0.183	0.997	0.874	0.182	0.997	0.884	0.181	0.997
	ovhd	0.384	1.00	0.874	0.381	1.00	0.834	0.372	1.00	0.837	0.380	1.00	0.830

Table 5: Exhaustive results of our experiments for the two naive approaches and `iCi`

lower false positive rate, while also offering reasonable runtime performance. In our evaluation, `iCi` achieves an f-score of `0.947` in the worst case, even at high optimization levels with many tail calls. We release our `iCi` prototype, as well as the oracle we developed for our evaluation, as open source.

REFERENCES

[1] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Sec'16*.

[2] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *EuroS&P'17*.

[3] Dennis Andriesse, Victor van der Veen, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proc. CCS'15*. ACM, Denver, CO, USA.

[4] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. 2001. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*.

[5] Bryan Roger Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *IJHPCA* 14, 4 (2000), 317–329.

[6] Franck de Goër, Christopher Ferreira, and Laurent Mounier. 2017. scat: Learning from a single execution of a binary. In *IEEE SANER 2017*. 492–496.

[7] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proc. USENIX Sec'13*.

[8] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI'05*. 190–200.

[9] Hanan Lutfiyya, Janice Singer, and Darlene A. Stewart (Eds.). 2004. *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, October 5-7, 2004, Markham, Ontario, Canada*. IBM.

[10] Yasuhiko Minamide. 2003. Selective Tail Call Elimination. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. 153–170.

[11] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *Proc.NDSS'15*.

[12] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *Proc. S&P'15*.

[13] Erik Putrycz. 2004. Using trace analysis for improving performance in COTS systems. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, October 5-7, 2004, Markham, Ontario, Canada*. 68–80.

[14] Michel Schinz and Martin Odersky. 2001. Tail call elimination on the Java Virtual Machine. *Electr. Notes Theor. Comput. Sci.* 59, 1 (2001), 158–171.

[15] Tomás Tauber, Xuan Bi, Zhiyuan Shi, Weixin Zhang, Huang Li, Zhenrui Zhang, and Bruno C. d. S. Oliveira. 2015. Memory-Efficient Tail Calls in the JVM with Imperative Functional Objects. In *APLAS*. 11–28.

[16] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *IEEE S&P'16*. 934–953.

[17] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity. In *Proc. NDSS'15*.

[18] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)*.

[19] Mingwei Zhang and R. Sekar. 2015. Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-World ROP Attacks. In *Proc. ACSAC'15*. 91–100.

[20] Yang Zhang, Xiaoshan Sun, Yi Deng, Liang Cheng, Shuke Zeng, Yu Fu, and Dengguo Feng. 2015. Improving Accuracy of Static Integer Overflow Detection in Binary. In *Proc.RAID'15*.