Adaptive Call-site Sensitive Control Flow Integrity

Mustakimur Khandaker* Abu Naser* Wenqing Liu* Zhi Wang* Yajin Zhou† Yueqiang Cheng‡
* Deparment of Computer Science, Florida State University, Tallahassee, USA
Email: {mrk15e, an16e, wl16c}@my.fsu.edu, zwang@cs.fsu.edu
† School of Computer Science, Zhejiang University, Hangzhou, China
Email: yajin_zhou@zju.edu.cn
‡ Baidu X-lab, Sunnyvale, USA
Email: chengyueqiang@baidu.com

***Abstract*—Low-level languages like C/C++ are widely used in various applications for their performance and flexibility. Unfortunately, these languages are prone to memory corruption vulnerabilities, leading to control-flow hijacking attacks. Control flow integrity (CFI) is a general principle to enforce run-time control flow of a program to a pre-computed control-flow graph (CFG). While the traditional context-insensitive CFI falls short in protecting critical control transfers, recent context-sensitive CFI research shows promising improvements but has various limitations.**

We present Control Flow Integrity with Look Back (CFI-LB), a call-site sensitive CFI in which a conventional source-target control transfer is strengthened by a look back into its call-sites (return addresses). CFI-LB features the adaptive call-site sensitivity in which each indirect call has its own level of sensitivity and the multi-scope CFG to improve the security even if a precise context-sensitive static CFG is not available, especially for large programs such as `GCC` and `NGINX`. One of the CFGs is constructed by our localized concolic execution, which significantly extends the dynamic CFG with very low false positives. In addition, CFI-LB is the first CFI system *explicitly* designed to protect its reference monitors from race conditions. We have built a prototype of CFI-LB. The evaluation with SPEC CPU2006 benchmarks and NGINX indicates that CFI-LB has a low-performance overhead (less than 5% on average for the full protection) while increasing the security.

I. Introduction

Computer software has become increasingly complicated. For example, the Linux kernel, first released in 1991, now has more than 20 million lines of source code (version 4.19.1). Such complexity unavoidably leads to more and more vulnerabilities [49], many of which can be exploited to execute arbitrary code or escalate the privilege. Existing deployed defenses such as data-execution prevention (DEP) and address-space layout randomization (ASLR) make exploits more challenging but can still be bypassed by code-reuse attacks [46], information leaks, heap sprays, etc.

Control-flow integrity (CFI) is an effective defense against most control-flow hijacking attacks [2]. It employs in-line reference monitors to enforce that the run-time control flow of a process must follow its control-flow graph (CFG), a static graph representing the program's legitimate control transfers. Control flow can be changed by direct branches (i.e., direct calls/jumps) and indirect branches (i.e., indirect calls/jumps and returns). Direct branches are hard-coded in the program and thus cannot be manipulated by attackers under DEP; indirect branches load the program counter with a code pointer from a register or a memory location. Consequently, they can be exploited by manipulating the in-memory data, and thus CFI needs to protect their integrity. Since its introduction, there has been a long stream of research in CFI to improve its performance [53], [54], increase its precision [18], [48], [50], and protect other mechanisms [36], [39], [3].) Nevertheless, recent research puts the security of CFI systems into serious question [6], [16], [21], [23].

The security of a CFI system is determined mainly by three factors: the actual CFG, the CFG construction algorithm, and the enforcement mechanism. The actual CFG is determined by the program structure itself. Some indirect branches have a large set of valid targets. In this case, even a precise CFI system could still be bypassed by taking advantage of these valid targets [6]. The CFG construction algorithm determines how close the computed CFG is to the actual CFG. CFGs are often computed by the static points-to analysis, a well-known NP-hard problem. Existing points-to analysis algorithms are designed to be conservative; consequently, illegitimate control transfers may be included in the resulting CFG and incorrectly allowed by the CFI system [21]. The enforcement mechanism can also introduce insecurity, mainly due to the trade-off between security and performance. For example, some CFI systems enforce an oversimplified CFG [47], [53], allowing them to be subverted [16], [23].

Context sensitivity is an effective way to improve the security of CFI systems [18], [48] because it takes the past execution into consideration when validating the next target of an indirect branch. Existing context-sensitive CFI (CS-CFI) systems rely on the hardware support, such as Intel last branch record (LBR) and processor tracing (PT), to obtain the recent execution path. These hardware features provide rich context information about both direct and indirect branches. However, Intel LBR and PT can only be accessed in the kernel mode; accordingly, these systems must change the kernel. This not only increases the complexity of the design but also incurs high performance overhead. This overhead can be partially addressed by enforcing CS-CFI at a limited number of selected locations, such as the syscall entrance [48], or by offloading the run-time check to a separate CPU core [18]. However, the former can only protect a small part of the program, and the rest of the program may still be compromised (e.g., to leak the private key of a web server); the latter reduces the number of usable CPU cores.

In this paper, we aim at improving the security and effectiveness of CFI with CFI-LB, *control-flow integrity with look-back*. CFI-LB is a CS-CFI system that uses call-sites as the context. Specifically, to validate the target of an indirect branch, CFI-LB obtains the return addresses from the safe shadow stack and only permits the control transfer if the target is valid for that sequence of callers. This is essentially a call-site sensitive CFI system. While deeper contexts (i.e., more call sites) provide better security, they often incur higher overhead, making the protection less useful. To address that, CFI-LB employs the following strategies: **first**, we observe that not all indirect branches require a deep context. In most cases, a single return address can provide sufficient constraint on the valid targets; while other cases require a deeper context for better security. As such, CFI-LB features the adaptive context sensitivity that allows each indirect branch to decide its own level of call sites to check. **Second**, CFI-LB features the multi-scope CFG to overcome the insecurity caused by imprecise/coarse-grained CFGs. Specifically, we compute three CFGs for every program – a dynamic CFG from dynamic profiling, a concolic CFG from localized concolic execution, and a static CFG from the static points-to analysis algorithm. The concolic CFG is derived from the dynamic CFG, but significantly expands it by exploring multiple paths at the point of interest with a concolic execution engine. The dynamic and concolic CFGs are precise but might be incomplete; while the static CFG is complete but may contain extraneous control transfers. At run-time, we apply different policies to these CFGs: control transfers within the dynamic and concolic CFGs are trusted and allowed by default, and these within the static CFG are allowed but recorded for further off-line verification. With the multi-scope CFG, CFI-LB is still secure even though a precise context-sensitive static CFG is not available. **Third**, an often overlooked pitfall of all the existing CFI systems is race conditions against CFI's inline reference monitors in multi-threaded processes. Many CFI systems employ reference monitors that save intermediate states in the memory (either by design or due to register spilling [1], [4], [34]). These states can be manipulated by another benign-but-vulnerable thread under attack. A secure CFI system must be "modeled against *arbitrary read/write at arbitrary times*" [35]. CFI-LB is the first CFI system that is *explicitly* designed to guarantee the atomicity of its reference monitors by leveraging the widely-available hardware transactional memory support (Intel TSX [28]). We have built a prototype of CFI-LB. In addition, we propose an equation to universally quantify the security of CFI systems. Our security and performance evaluation demonstrates that CFI-LB can significantly improve the security of CFI systems without causing high performance overhead (less than 5% on average for the full protection of indirect branches and returns and Intel TSX support).

In summary, this paper makes the following contribution:

- We propose the design of CFI-LB, a call-site sensitive CFI system featuring adaptive context sensitivity and multi-scope CFG to provide the balanced security and improve the security even if a precise context-sensitive CFG is not available (as is the reality).
- CFI-LB is the first CFI system explicitly designed to protect the integrity of its reference monitors against race conditions. In addition, our localized concolic execution can significantly extend the dynamic CFG with low false positives.
- We propose a universal quantitative metric to measure and compare the security of both context-insensitive and context-sensitive CFI systems.
- We have built a prototype of CFI-LB and extensively evaluated its security and performance.

II. Call-site Sensitive CFI

In this section, we first propose a generic metric to quantify the security of CFI systems and then present the notion of call-site sensitive CFI with a concrete example.

A. *Quantifying Context-Sensitive CFI*

CFI protects a process from control-flow hijacking attacks by confining the possible targets of an indirect branch to these in the program's CFG. As previously mentioned, the security of a CFI system is determined by the following three factors: the actual CFG, the CFG construction algorithm, and the enforcement mechanism. The first factor is decided by the program structure itself. For example, syscalls in the kernel are dispatched through a large table of syscall handlers, indexed by the syscall number. Recent versions of the Linux kernel for x86-64 contain more than 320 syscalls. This particular indirect call hence has a very large legitimate target set. Target sets can also be affected by the precision of the CFG construction algorithm (the second factor), i.e., points-to analysis. Points-to analysis is a static analysis to calculate the variables or data structures a pointer can point to. Precise points-to analysis algorithms rely on context sensitivity (e.g., call-site sensitivity) to improve precision. Unfortunately, such algorithms often do not scale well. Few can handle the code size of real-world programs such as `Apache` and `GCC`, and even fewer are publicly available and regularly maintained (we report our own experience later). The last factor is the CFI enforcement mechanism. CFI inserts online reference monitors into the target program to enforce its policies. A precise CFI enforcement requires to perform more thorough check of the run-time states and thus may lead to higher performance overhead. As such, some CFI systems trade precision and security for performance, making them bypassable. Existing CFI systems have applied a wide spectrum of designs in these three factors. An effective measurement of the security of CFI systems should take all these three aspects into consideration.

In a recent survey of (context-insensitive) CFI systems [3], Burow et al. propose to use the following equation to quantify the security of CFI systems:

$$QS_{CFI} = EC \times \frac{1}{LC} \tag{1}$$

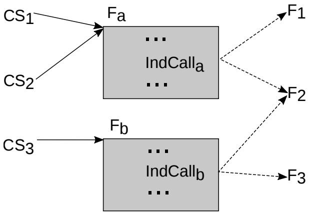

Fig. 1. An example of equivalence classes. CS_x represents a call site; IndCall represents an indirect call instruction; and F_x represents a function.

EC is the total number of the equivalence classes and LC is the largest size of the equivalence classes. An equivalence class (EC) is a group of targets that a CFI system cannot distinguish. Fig. 1 gives an example of ECs: function F_a and F_b both contain an indirect call, $IndCall_a$ and $IndCall_b$, respectively. The former targets function F_1 and F_2, and the latter targets F_2 and F_3. F_a is called by call site CS_1 and CS_2 while F_b is called by CS_3. A traditional CFI system validates the targets without considering the contexts. This creates at most two ECs ($\{F_1, F_2\}$ and $\{F_2, F_3\}$). However, the CFI enforcement may merge these two classes. For example, the original CFI system [2] inserts a label at each target and enforces CFI by checking the label. This requires these two ECs be merged because each target function can only bear one label; the label of F_2 thus has to be shared by F_1 and F_3. Some CFI systems trade security for performance by assuming every indirect call can target any address-taken function. As such, they have a single large EC. In summary, EC reflects the precision of both the CFG and the enforcement mechanism. Eq. 1 is an effective measurement of the security of context-insensitive CFI systems, but it is not as effective for CS-CFI systems.

Call-site sensitive CFI takes the call path into consideration when enforcing the CFI policy. This leads to more, smaller ECs. For example, assume there are four valid execution paths in Fig. 1: ($CS_1 \rightarrow F_a \rightarrow IndCall_a \rightarrow F_1$), ($CS_2 \rightarrow F_a \rightarrow IndCall_a \rightarrow F_2$), ($CS_3 \rightarrow F_b \rightarrow IndCall_b \rightarrow F_2$), and ($CS_3 \rightarrow F_b \rightarrow IndCall_b \rightarrow F_3$). These four paths are divided into three ECs with a maximum size of 2 – the first two paths are in its own class separately, and the last two are in one class together. However, different context-sensitive CFI systems may use different contexts, leading to different numbers of ECs in the CFG. Moreover, the number of ECs could increase exponentially while the maximum EC size changes at a much slower rate (see an example in Section III-B). Because of these reasons, Eq. 1 is not suitable for comparing CS-CFI systems. To address that, we propose to use the following equation to measure the security of *all* CFI systems:

$$QS_{CFI} = AVG_{EC} \times LC \tag{2}$$

AVG_{EC} is the average size of all the ECs, and LC is still the size of the largest EC. In Eq. 2, the larger QS_{CFI} is, the less secure. A useful feature of Eq. 2 is that QS_{CFI} now has a theoretical limit of 1, in which every target can be individually distinguished and validated.

B. Call-site Sensitive CFI

```
 1  typedef int (*Handler)(char *);
 2  int proceed(Handler handler, char *root_path)
 3  {
 4    ...
 5    return handler(root_path);
 6  }
 7
 8  void auth()
 9  {
10    char *user_name;
11    char *passwd;
12    char id[80];
13    int attempt = 5;
14    Handler handler;
15
16    while (attempt > 0) {
17      handler = &on_failure;
18      username = passwd = null;
19
20      scanf("%ms", &username);
21      scanf("%ms", &password);
22
23      passwd = salt_passwd(username, passwd);
24      sprintf(id, "%s;%s", username, passwd);
25
26      if (is_admin(id)) {
27        handler = &on_admin;
28        proceed(handler, user_home_dir);
29      } else {
30        proceed(handler, "/tmp");
31      }
32      attempt--;
33
34      //clear passwd, free user_name, passwd
35      ...
36    }
37  }
```

Fig. 2. An example for call-site sensitive CFI

CFI-LB enforces a call-site sensitive CFI policy in which the targets of an indirect branch are validated in the context of call-sites (i.e., return addresses on the stack). By doing so, we can partition equivalence classes into finer-grained sets, reducing the average EC size and improving the overall security (Eq. 2).

Fig. 2 illustrates the benefits of call-site sensitivity, in which the `auth` function authenticates the user and calls the `proceed` function with a function pointer decided by the results of `auth`. In the context-insensitive CFI, the indirect call at `L5` can legitimately transfer to both `on_admin` and `on_failure`. That is, an attacker can execute `on_admin` even if the password authentication fails by exploiting the buffer overflow at `L24` to overwrite `handler`. By nature, even a precise context-insensitive CFI will fail to provide meaningful protection for this program.

With CFI-LB, we take the call sites into account when validating the targets of `handler` at `L5`. Specifically, `proceed` has two call sites at `L28` and `L30`. The valid targets of `L5` thus can be presented by tuple (`L28`, `L5`, `on_admin`) and (`L30`, `L5`, `on_failure`). To validate the targets at `L5`, we retrieve the return address from the stack, combine it with the location of the indirect call (L5) and the target address, and check whether the formed tuple is valid or not. Consequently, we can prevent the aforementioned attack. Note that the attacker cannot overwrite the return address of proceed because it is pushed to the stack after the overflow, and we employ a secure shadow stack to protect return addresses. Any modifications to them will trigger an exception. In other words, the call sites used in the validation are trusted.

What we just described is technically a one call-site sensitive CFI. Sometimes more call sites need to be examined to provide a finer-grained context, for example, if the indirect call in L5 is wrapped in another function and proceed calls this function instead of directly calling handler. In this case, a two call-site sensitive CFI-LB can recover the lost precision. As expected, call-site sensitivity may fall short in some cases even though it is overall a strong protection. For example, it is possible to merge the two calls of proceed in Fig. 2 by introducing a new local variable to store the arguments of root_path. Now, proceed has only a single call site. This prevents CFI-LB from distinguishing the two targets of handler. To further improve the precision, we need to take the executed branches into account, i.e., which branch of is_admin is executed. An Intel-PT based CFI can theoretically provide this kind of protection [18]. However, the CFG is the limitation here – a points-to analysis with that level of precision (context- and path-sensitive) is hardly scalable, and its public availability is virtually non-exist, especially for the C/C++ language.

III. System Design

In this section, we first describe how we enforce one call-site sensitivity in CFI-LB and then describe two unique features of CFI-LB, adaptive call-site sensitivity and multi-scope CFG, to make our design more scalable and secure.

Assumptions: like all other CFI systems, we assume that the code integrity of the target program is protected by W⊕ X, a common protection in all commodity operating systems. A secure CFI system has to protect both returns and indirect calls (or jumps). We assume that return addresses are protected by a secure shadow stack [11], [29], which saves a copy of return addresses to a protected stack and verifies that they have not been changed before returning. Our prototype uses CPI's SafeStack [31] for this purpose. SafeStack separates return addresses and others safe data into a separate safe stack. It can protect return addresses from being compromised, similar to the shadow stack.

A. *Enforcing One Call-site Sensitivity*

To enable its protection, CFI-LB instruments the program to insert inline reference monitors that protect the program's run-time control flow. We describe this process in two steps, compiling-time instrumentation and run-time verification.

Compiling-time instrumentation: Fig. 3 shows the CFG for the example program in Fig. 2. There is one indirect call (L5) (and four returns) that needs protection. To protect indirect calls, CFI-LB uses a hash-table based set membership test.

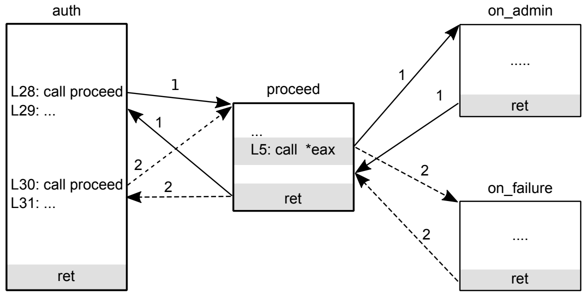

Fig. 3. CFG for the example in Fig. 2

For one call-site sensitivity, the set consists of tuples of the following format: (return site, indirect call, target). Therefore, the set for Fig. 3 contains (L29, L5, on_admin) and (L31, L5, on_failure). We instrument the program to initialize the hash table with these two entries before executing the main function. Our current prototype uses a simple hash function to calculate the hash index: xor the three members of each tuple and mod the set size. We practically adjust the set size to reduce hash conflicts. Our experience with large programs, such as SPEC CPU2006, shows that this simple hash function is fast and effective. For example, the hash table for 403.gcc has 1,541 entries with only 74 conflicts. More advanced hash functions can easily be adopted if necessary. The hash table is write-protected after the initialization.

We further instrument the program to insert a reference monitor before each indirect call. In our prototype, we instrument the program at the source code level. The hash table search function is implemented as a function but inlined and specialized at each instrumentation point (e.g., L5 in Fig. 3). Inlining is used to avoid introducing an artificial function (i.e., the validation function) that can legitimately return to all the locations before the indirect calls.

We use the hash-table based enforcement because it will not introduce imprecision since it does not merge the target sets of different indirect calls, and it has near constant performance. As we will show later, the size of target sets can increase exponentially when the level of call-site sensitivity increases. Hash table can provide the consistent performance.

Run-time verification: at run-time, the reference monitor retrieves the target from the register/memory and return addresses from the shadow stack, hash them with the location of the indirect call (hard-coded in the reference monitor), and search the hash table for validation.

Atomicity of reference monitors: an often overlooked attack vector against CFI is the race condition – if a reference monitor saves its intermediate states in the memory, the attacker can compromise these states by exploiting vulnerabilities in another thread. Even though the window of vulnerability is narrow, reference monitors are called frequently enough for this to be a concern. A simple defense against that is to rewrite reference monitors to only use the registers. This method works because each CPU core has its own set of registers, and one core cannot change another core's registers. However, it is difficult to implement since it requires hand-coding reference monitors in assembly instructions. Most existing CFI systems rely on the compiler for register allocation (e.g., systems implemented in the LLVM IR [1], [4], [34]). Their internal states might be spilled to the stack by the compiler, making them potentially vulnerable.

To address that, CFI-LB encapsulates its reference monitors in the hardware transactional memory based on Intel TSX [28]. Intel TSX provides two software interfaces, hardware lock elision (HLE) and restricted transactional memory (RTM). CFI-LB uses the RTM interface, which consists of three new instructions, `xbegin`, `xend`, and `xabort`, to start, end, and abort a transaction, respectively. When a transaction is started, the CPU records all the memory accesses by the transaction and monitors the system for race conditions (i.e., read-write and write-write conflicts). If a race condition is detected, the transaction is aborted and its changes are rolled back; otherwise, it is committed to the memory. When a transaction fails, the CPU returns the error conditions in the `eax` register. One of the return status is conflict-detected (`_XABORT_CONFLICT`). However, we cannot simply report a conflict as an attack because a transaction can fail without real data conflicts – Intel TSX detects data conflicts at the granularity of a cache line (64 bytes) [28]. False sharing in the cache line or cache conflicts can all cause transactions to fail. In our experiment with the NGINX server, we found that transactions have a chance of 0.003% to fail even without race conditions. To address that, we retry the transaction multiple times before reporting a failure. This design does not weaken the security because the verification can not be compromised as long as the transaction succeeds. To the best of our knowledge, CFI-LB is the first CFI system that is explicitly designed to guarantee the atomicity of its reference monitors.

B. Adaptive Call-site Sensitivity

As specified in Eq. 2, the security of CFI-LB is determined by the average EC (equivalence class) size and the largest EC size. In the following, we use the `GCC` benchmark in SPEC CPU2006 as an example to demonstrate the impact of different levels [1] of call-site sensitivity to these two parameters. Our measurement of other benchmarks shows a similar trend (Section IV). We then describe the first feature of CFI-LB, adaptive call-site sensitivity, to balance security and performance.

TABLE I
NUMBER OF ECS AND AVERAGE EC SIZE FOR DIFFERENT LEVELS OF SENSITIVITY.

Sensitivity	Call-site(0)	Call-site(1)	Call-site(2)	Call-site(3)
# of ECs	220	795	2763	6463
AVG_{EC}	2.7	1.53	1.38	1.33

Average EC size: the goal of call-site sensitivity is to limit the choices that an attacker has in compromising the control flow by reducing EC sizes. Generally speaking, more levels of call-site sensitivity lead to smaller average EC sizes. Table I shows the result for the `GCC` benchmark [2]. For example, with the three call-site sensitivity, the average EC size decreases to 1.33 but the number of ECs increases more dramatically ($29\times$ of the context-insensitive CFI). This implies that a constant-time membership test algorithm is essential to the performance of CFI-LB. Nevertheless, increasing the level of sensitivity can still harm the performance because it needs more memory accesses to retrieve the context.

TABLE II
EC DISTRIBUTION OVER SIZES OF THE TARGET SETS

Sizes	Call-site(0)	Call-site(1)	Call-site(2)	Call-site(3)
1	149	674	2372	5423
2	27	54	223	667
3	10	18	48	172
4	7	17	35	70
5-10	20	26	56	93
11-20	4	4	28	37
21-40	2	1	0	0
54	1	1	1	1
Total	220	795	2763	6463

Largest EC size: the security of a CFI system also relies on the largest EC size. When increasing the call-site sensitivity, larger ECs break down into smaller ECs. This can potentially reduce the largest EC size. Table II shows the distribution of EC sizes. For example, with one call-site sensitivity, there are 674 ECs having a single target and 26 ECs having between 5 and 10 targets. When we increase the sensitivity to three, these numbers increase to $5,423$ and 93, respectively.

Interestingly, the maximum EC size for every case remains the same (54). This problematic indirect call is located in Function `get_insn_template`, which uses a function pointer to generate templates for different instructions. The reason that this indirect call defies call-site sensitivity is because its only caller, `final_scan_insn`, is a recursive function. As such, the caller can appear many times on the call stack, and no level of call-site sensitivity can increase the precision for this call. We could theoretically merge duplicated return addresses in the context. Such an implementation is more complicated and has higher overhead. In summary, using a high-level of sensitivity across the program has the following issues: higher performance overhead, explosion of the number of ECs, and cases that it does not improve security.

Adaptive call-site sensitivity: to address these problems, CFI-LB features adaptive call-site sensitivity, in which the level of sensitivity is decided independently for each indirect call. The goal is to reduce the average EC size while avoiding the explosion of the number of ECs. The algorithm is summarized as the following:

1) Repeat the following for the maximum level $\mathbf{n}$ from 0 and 3: for each indirect call, calculate AVG_{EC} for each level of sensitivity from 0 to $\mathbf{n}$, and select the smallest level that has the smallest AVG_{EC} for this indirect call.

[1] The level of sensitivity counts how many levels of return addresses to use.

[2] We use the CFG generated through dynamic profiling from the reference inputs of the SPEC CPU2006 benchmark to avoid the imprecision of CFG generation algorithms.

TABLE III
DISTRIBUTION OF ADAPTIVE CALL-SITE SENSITIVITY. NOTE THAT THE LARGEST INDIRECT CALL IS CORRECTLY ASSIGNED TO LEVEL 0 SINCE NO ADDITIONAL LEVEL WILL REDUCE ITS TARGET SIZE.

Call-site Depth	Max. target Set Size	# of Indirect Calls	# of ECs	Avg. EC Size
0	54	161	161	1.48
1	20	23	262	1.37
2	17	36	787	1.50
Total		220	1210	1.47

2) Pick the maximum level $\mathbf{n}$ that has the best security according to Eq. 2. $\mathbf{n}$ is the maximum level of sensitivity for the whole program.

The results of applying this algorithm to `403.gcc` is shown in Table III. The maximum level is 2 with an average EC size of 1.47. The total EC number is $1,210$, less than half of the case if two call-site sensitivity is applied uniformly. Note that CFI-LB's hash-table based CFI enforcement can easily support adaptive sensitivity by assuming the missing call-sites to be zero in lower levels.

C. Multi-scope CFG

CFI-LB's second feature, multi-scope CFG, focuses on improving the security even if a scalable context-sensitive points-to analysis algorithm is, unfortunately, not available for C/C++ programs (the current state of the art in the points-to analysis). We combine different CFGs together and apply both on-line and off-line validation of control transfers.

The need for multi-scope CFG: there are two ways to generate CFGs: dynamic analysis and static analysis. Dynamic analysis runs the program under a dynamic binary instrumentation (DBI) tool to record an execution history of the program and construct the CFG from it. For brevity, we call the CFG created in this way a dynamic CFG. A dynamic CFG has no false positives, i.e., all the control transfers are valid. However, it suffers from false negatives, i.e., the CFG is incomplete because dynamic analysis cannot cover all the code paths.

Static analysis uses points-to analysis to calculate the CFG from either the source code or the program binary. Traditionally, points-to analysis is used by compilers to optimize programs. The produced CFG is an over-approximation of the real CFG, i.e., the CFG has false positives (extra control transfers) but no false negatives (missing control transfers). One way to improve the precision of points-to analysis is context sensitivity. Call-site sensitivity is often used for procedural programming languages like C/C++. Unfortunately, precise points-to algorithms do not scale well, especially for complicated programs like `GCC`. The availability of such algorithms is even worse. The best choice at the time of writing seems to be the DSA algorithm, which has not been maintained for a long time and contains known algorithmic errors [32], [33]. Existing CFI systems often use an imprecise CFG, such as all address-taken functions [47] or the type-based CFG [1].

Given this reality, CFI-LB proposes the multi-scope CFG to improve security despite the imprecision in the points-to analysis: a security mechanism like CFI should have no false positives and limited false negatives to be secure and usable [3]. Note that a false positive in the CFG leads to false negatives in the CFI (no alarm raised where it should be), and vise versa. Therefore, the overall CFG we use should have no false negatives and limited false positives. CFL-LB's multi-scope CFG consists of three CFGs: a dynamic CFG generated by dynamic analysis, a static CFG generated by a scalable but conservative points-to analysis, and a concolic CFG generated by our localized concolic execution. The relation of these three CFGs are shown in Fig. 4.

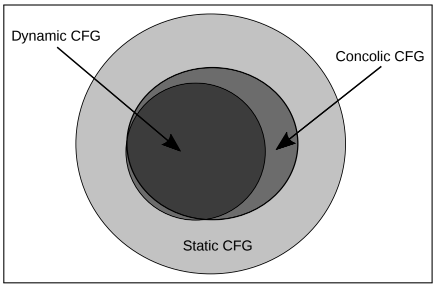

Fig. 4. Multi-scope CFG in CFI-LB

On-line and off-line verification: after generating those CFGs, CFI-LB applies both on-line and off-line verification of indirect call targets. Specifically, any run-time control transfer within the dynamic CFG is allowed because the dynamic CFG contains only valid control transfers. Our experiments show that the concolic CFG has a very low false positive rate. As such, we also allow any run-time control transfer within the concolic CFG. However, the static CFG may cause false negatives in the CFI. We allow a control transfer in this CFG (but outside the dynamic and concolic CFGs) to proceed at run-time but record the target and its context (e.g., the call stack) for the off-line analysis. If the analysis proves that the control transfer is legitimate, it will be added to the dynamic CFG for future use. Any control transfer outside the static CFG is a true violation and will be immediately blocked. The rationale of this design is that only few run-time control transfers require off-line verification because of the effectiveness of the dynamic and concolic CFGs; we can gradually expand the dynamic CFG, and eventually any run-time control transfers out of the dynamic and concolic CFGs become suspicious.

To support multi-scope CFG, we mark each hash table entry with its scope and record the run-time context as indicated by the scope. The context we record includes the call stack and the arguments to the target function. We can use this information to validate the control transfer. In the following, we present our method to generate the concolic CFG.

D. Localized Concolic Execution

The goal of localized concolic execution is to extend the dynamic CFG with very few false positives so that CFI-LB

[3] A typical counter-example is UAC of Window Vista that trained users to ignore and click through any access control questions.

can trust them without further validation.

Overview: concolic execution, as its name indicates, is a combination of concrete and symbolic execution. It maintains the symbolic relations between program variables while concretely executing the program. Concolic execution is often used to improve path coverage in software testing: when the concolic execution engine finds a conditional branch, it adds the branch's condition to the path condition. The satisfiability of the path condition determines whether a path is feasible or not. To explore an alternative path, the engine tries to solve the related path condition with a constraint solver, such as Z3 [17]. If the solver proves the path condition is satisfiable, it provides a solution to the related symbolic variables that will lead the execution to that path in a concrete execution.

Theoretically, we could use concolic execution to explore all the paths of the program and derive a complete CFG. Unfortunately, concolic execution does not scale well because of the path explosion problem, in which the number of feasible paths grows exponentially to the number of conditional branches and can even be infinite if the program has unbounded loops. Our target programs are too complicated to be fully explored. For example, `403.gcc` has about 1 million instructions.

CFI-LB addresses this challenge by limiting the scope of concolic execution with additional heuristics to further improve its scalability. Because the generated CFG is used in the call-site sensitive CFI, we must generate, for each indirect call, both its target set and the associated contexts. CFI-LB's concolic execution is localized to the individual indirect call site. Specifically, at each indirect call site, it searches backward in the program's CFG for the callers of this indirect calls and concolically executes these callers until the execution reaches the indirect call. A list of the targets and their associated contexts will be returned at the indirect call site. The intuition behind this approach is that the function pointer used by the indirect call likely is assigned or indexed by its callers. By exploring more code paths in the callers, we can potentially assign other legitimate values to this function pointer. In the following, we will describe each step in detail.

Step I: recording initial program states: CFI-LB's localized concolic execution is binary based. As previously mentioned, concolic execution runs the program both concretely and symbolically. It is necessary to provide the valid inputs to the starting function so that the concrete execution can proceed without causing exceptions. Our target programs are often very complicated. It is generally unfeasible to achieve this programmatically. To this end, we execute the program under a dynamic binary instrumentation (DBI) tool with some valid inputs (e.g., a well-formed C program for GCC) and record a complete execution history. At the entry point of an interested function, we reconstruct the function's arguments, the global variables, the registers, and the heap from the execution history and take a snapshot of them. This snapshot provides a set of valid inputs for the concolic execution. We use Intel Pin as the DBI tool in our prototype [30].

Step II: locating starting functions: after recording the execution history, our system needs to find the callers of the interested indirect call site as the starting function for the concolic execution. While recording the execution history, the DBI tool also generates a dynamic CFG for the program. Technically, we can search this CFG for the callers of this indirect call and start concolically executing them. However, this approach does not work well – sometimes the indirect call that we are interested in is not even executed by the DBI tool because of the limited coverage of the initial input. To address that, we extract from the program binary a CFG that contains only direct calls, and further extend this CFG by the indirect control transfers from the dynamic CFG. We then locate the callers of the interested indirect call in this CFG. If a located caller has been executed by the DBI tool, we add this caller into a work list for the concolic execution. We make sure that the depth of the callers is larger than or equal to the depth of this indirect call's sensitivity.

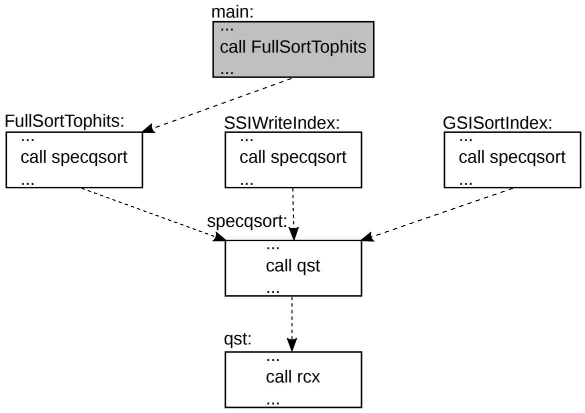

Fig. 5. Locate the callers of an indirect call (in `456.hmmer` benchmark)

Fig. 5 illustrates how this works. Specifically, the `qst` function, which implements quick sort, contains an indirect call through the `rcx` register. `qst` has a single caller, `specqsort`. Unfortunately, `specqsort` has not been executed by the DBI tool; we then have to search backwards further. `specqsort` in turn has three callers, but none of them have been executed by the DBI tool either. Searching further back, we find that one of `specqsort`'s callers, `FullSortTophits`, can be called by the `main` function, which certainly has been executed by the DBI tool. Therefore, we add `main` as a starting function and record the path from `main` to `qst` (`main` → `FullSortTophits` → `specqsort` → `qst`). This information is used by the concolic engine to limit the depth of exploration (more details in the following). Note that even though the `main` function includes a call to `FullSortTophits`, `FullSortTophits` may not have been executed by the DBI tool if it is guarded by an unsatisfiable branch condition during the initial execution. Our concolic engine can solve the related path condition and execute the function. If we just search for starting functions in the dynamic CFG generated by the DBI tool, we cannot find any executed callers for `qst`. Therefore, by searching for starting functions in the combine CFG, we can discover more targets of an indirect call.

Step III: concolically executing code: after discovering the starting functions, we start executing them one by one concolically. Specifically, we use the aforementioned snapshot as the initial inputs to the starting function and execute the program concretely while maintaining the relation between symbolic variables. In particular, we maintain a set of path conditions. If a new conditional branch is found, we add the condition (or its inverse depending on which branch the concrete execution takes) to the current path condition. After every emulation, we iterate through the generated path conditions and check whether a path has been taken or not. For each path not-taken, we then query a SMT solver to check whether its path condition can be satisfied. If so, the solver will return a model that will execute the new path. This process repeats until all the paths have been explored. When the concolic engine reaches the indirect call site, we retrieve the values stored in the register or memory referenced by the indirect call and add it and its context into the target set of the indirect call.

Variable symbolization: when concolic execution is used to explore a whole program, we just need to symbolize the external inputs, such as the command line arguments and files. In CFI-LB, we only concolically execute a part of the program. Therefore, we symbolize all the function arguments, global variables, and external inputs (e.g., files). Specifically, if a variable is a scalar (e.g., an integer), we create a new symbolic variable for it; if a variable is a pointer to a data structure, we keep the pointer's concrete value (i.e., the address of the structure in the snapshot) but symbolize the data structure it points to; if a variable is a function pointer, we do not symbolize it otherwise it can point to arbitrary memory or whatever the solver produces. The following example contains all these three cases:

```
void specqsort(char* base, int n, int size, int
                    (*compar)());
```

By symbolizing all the inputs external to the starting function, we essentially assume that they can take any value. Consequently, we over-approximate the results (false positives in the CFG) because some values provided by the solver may not be possible in a normal execution.

Optimization: our concolic engine uses a few measures to improve the scalability. For example, we limit each loop to only five iterations if the loop is unbounded. One of the most significant challenges to scalability is function calls. The instructions to simulate can cascade quickly when function calls nest. To address that, we simply ignore the function calls unrelated to the indirect call: we have mentioned that we maintain a path from the starting function to the indirect call. Functions on this path are concolically executed while functions not on the path are ignored. However, if any argument to an ignored function is symbolic, we symbolize the function's return value. This is another over-approximation since we assume the function's return can take any value if one of its arguments is symbolic.

Not all unrelated functions are ignored. Specifically, our execution engine can emulate more than 60 common `libc` functions. If an unrelated function is a libc function, we call the corresponding emulated function, which simulates the behaviors of the function. Most libc functions are straightforward to emulate. For example, we symbolize the related buffer for functions such as `fread`, `fgets`, and `getenv` since they accept inputs from the external resources. For functions that modify the memory (e.g., s̃trcat, `memcpy`, `strcpy`), we emulate their operations on the memory (e.g., to create a new buffer) and symbolize the resulting memory if any of the input is symbolic. Functions like `strlen` is interesting. As we have mentioned, CFI-LB symbolizes the content of a buffer but neither its address nor its length. To correctly simulate `strlen`, the symbolic engine needs to support variable length buffer, as well as to change the solver's memory model. Our experience with several publicly available symbolic engines shows that these features currently are not well supported. To address that, we just symbolize the return value of `strlen` if its input is a symbolic buffer. Despite these approximations, our experiments with large programs show that the computed target set contains relatively few false positives.

E. Prototype of CFI-LB

We have built a prototype of CFI-LB for the Intel x86-64 architecture. We wrote a tool for the Intel Pin DBI framework to record the program execution and implemented the localized concolic execution based on the Triton symbolic engine [43]. The instrumentation is implemented in the Clang's CodeGen module, a module after the source parsing but before the LLVM IR generation. We used LLVM's CFL-AA alias analysis algorithm to generate the static CFG. Overall, our prototype has 4500+ lines of C/C++ code and 500+ lines of python code.

IV. Evaluation

We present the evaluation of our prototype in this section. In particular, we aim at answering the following questions: *first*, whether adaptive call-site sensitivity improves the security of CFI as measured by Eq. 2 in Section II; *second*, whether the localized concolic execution can extend the dynamic CFG with few false positives; *third*, what's the performance overhead of our prototype in the different configurations.

A. Security Evaluation

To answer the first question, we need to quantitatively analyze the security of CFI-LB using Eq. 2.

Effectiveness of call-site sensitivity: intuitively, the security of CFI improves when both average and largest EC sizes decrease. Fig. 6 shows the impact of call-site sensitivity on the number of ECs, the average EC size, and the largest EC size. As the level of sensitivity increases, the average EC size generally drops, and the attacker has less choices of the legitimate targets. For example, at three call-site sensitivity, the average EC sizes of three benchmarks (perlbench, gcc, and gobmk) drop to less than half of the context-insensitive CFI. These benchmarks happen to be the three largest ones. Another benchmark, h264ref, drops to about 70%. However, the remaining three benchmarks, hmmer, bzip2 and sjeng, have the same average EC size as the context-insensitive CFI.

TABLE IV

EFFECTIVENESS OF ADAPTIVE CALL-SITE SENSITIVITY. THE TABLE SHOWS THE MAX CALL-SITE LEVEL, THE NUMBER OF INDIRECT CALLS IN EACH LEVEL, AND FOR EACH LEVEL, THE NUMBER OF ECS AND THE AVERAGE EC SIZE. THE LAST COLUMN SHOWS THE IMPROVEMENT OF CFI-LB OVER CONTEXT-INSENSITIVE CFI.

Benchmark	Lang	Max Level	# of IndCalls	# of ECs	AVG_{EC} Per Level	AVG_{EC}	LC	$QS_{CFI-LB}/QS_{CFI(0)}$
400.perlbench	C	3	62/8/3/7	62/30/114/492	1.02/1.0/1.21/2.77	2.28	115	1/2.4
401.bzip2	C	0	12	12	1.0	1.0	1	1
403.gcc	C	2	161/23/36	161/262/787	1.48/1.37/1.50	1.47	54	1/1.84
429.mcf	C	0	0	0	0	0	0	0
445.gobmk	C	2	36/15/12	36/39/98	16.86/9.44/12.76	12.86	427	1/2.4
456.hmmer	C	0	9	9	1.0	1.0	1	1
458.sjeng	C	0	1	1	6.0	6.0	6	1
462.libquantum	C	0	0	0	0	0	0	0
464.h264ref	C	3	68/2/4/1	68/20/48/12	1.5/1.05/1.15/1.25	1.31	2	1/6.4
471.omnetpp	C++	3	226/2/8/3	226/86/156/60	1.81/1.0/1.04/1.7	1.44	168	1/1.45
473.astar	C++	0	1	1	1.0	1.0	1	1
483.xalancbmk	C++	3	1960/25/30/48	1960/117/118/262	1.06/1.12/1.20/1.71	1.14	26	1/1.52
433.milc	C	0	1	1	2.0	2.0	2	1
444.namd	C++	0	12	12	1.0	1.0	1	1
447.dealII	C++	3	100/3/4/1	10/19/20/9	1.04/1.0/1.0/1.11	1.03	2	1/1.07
450.soplex	C++	0	56	56	1.0	1.0	1	1
453.povray	C++	2	40/3/9	40/10/33	1.6/4.2/2.12	2.12	9	1/1.06
470.lbm	C	0	0	0	0	0	0	0
482.sphinx3	C	0	0	0	0	0	0	0
NGINX	C	3	94/18/0/11	94/89/0/58	5.54/1.06/0.0/4.91	3.73	62	1/3.3

They happen to have the smallest code sizes. It seems that complex code benefits more from call-site sensitivity. Note that the average EC size for gobmk increases from two call-site sensitivity to three. This is possible if the total EC size increases faster than the number of ECs.

However, we found that the largest EC size in almost all the benchmarks do not meaningfully reduce, except h264ref (reduced to 20%). We further looked into the reasons of this. *First*, for gcc, the caller of this particular indirect call is a recursive function; thus its largest EC cannot be broken down regardless the level of sensitivity. *Second*, the largest EC of gobmk is related to an indirect call in the shapes_callback function, which handles different board shapes in a Go game. shapes_callback are called by a sequence of callers: matchpat → matchpat_loop → do_matchpat. The function pointer called by shapes_callback is actually defined in the caller of matchpat; therefore increasing the sensitivity to four will significantly reduce the largest EC size of gobmk. However, this might lead to the explosion of EC numbers. *Third*, perlbench cannot benefit from increasing the level of sensitivity because the related indirect call is called close to the main function. Using a higher level of sensitivity will go into the C run-time (executed before main) without improving security.

In short, call-site sensitivity improves the security of CFI systems since the average EC size decreases while the largest EC size remains the same or also decreases.

Effectiveness of adaptive call-site sensitivity: CFI-LB employs the adaptive call-site sensitivity to balance security, performance, and the number of ECs. As shown in Fig. 6, the number of ECs can increase quickly when the level of sensitivity increases. To address that, CFI-LB calculates, for each indirect call, the sensitivity level that leads to the minimal average EC size. Table IV shows the results of applying adaptiveness to these benchmarks, including SPECint2006, the C/C++ benchmarks of SPECfp 2006, and NGINX. In particular, Fig. 6 shows that increasing the level of sensitivity does not improve the security for hmmer, bzip2, and sjeng. Our adaptive algorithm correctly sets their max levels to 0. The same is applicable to astar, namd, and soplex. The algorithm can often assign level 0 to the majority of indirect calls. For example, $1,960$ indirect calls out of the $2,063$ ones for xalancbmk are assigned to level 0 with an average EC size of 1.06. Overall, the final average EC sizes are well under control except for gobmk and sjeng. The structure of either programs is not very amicable to call-site sensitivity. The last column shows the improvement of CFI-LB over the context-insensitive CFI as measured by Eq. 2. Note that a smaller QS indicates better security. As such, the best improvement is h264ref, while context-insensitive CFI can already provide sufficient protection for bzip2, hmmer, and several other benchmarks.

We also evaluated our prototype with NGINX. We built NGINX with OpenSSL, pcre, and zlib libraries, and used the NGINX testsuite as the standard inputs. The results are shown in Table IV as well. The largest EC has 62 targets and the average EC size is 3.73. Unlike the benchmarks in SPEC CPU2006, NGINX has 11 indirect calls located in the callback functions. For example, during the initialization, the OpenSSL library registers the OPENSSL_cleanup function to be called at the normal process termination (via atexit). OPENSSL_cleanup is thus a callback function called by an external module (i.e., glibc in this case). OPENSSL_cleanup contains an indirect call that usually calls ssl_library_stop to stop the SSL library. To protect this indirect call, CFI-LB has to limit the call sites within the NGINX program because the external module could be loaded

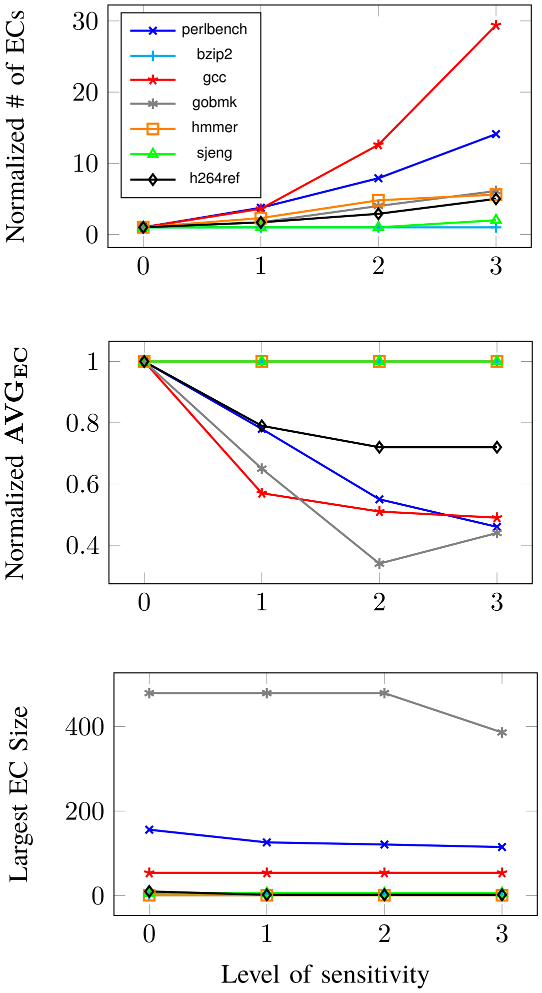

Fig. 6. The impact of call-site sensitivity on the number of ECs, average EC size, and the largest EC size for C benchmarks in Spec CPU2006. Level 0 represents the traditional context-insensitive CFI. The number of ECs and average EC size are normalized to fit in the figures. 429.mcf and 462.libquantum have no indirect calls exercised. The curve for C++ benchmarks are similar.

at a different location each time the program is run. To improve the protection of callback functions, we could compile and link these modules statically into the main program or update the CFGs at the run-time [38].

In short, adaptive call-site sensitivity can substantially improve the security of CFI while curbing increases in the number of ECs.

Effectiveness in preventing control flow hijacking: we used the RIPE benchmark [51] and three real-world vulnerabilities to test whether CFI-LB can detect and block control-flow hijacking attempts. These three vulnerabilities consist of two heap overflows that target a function pointer and a stack-based overflow that targets the return address. The former tests CFI-LB's forward-edge protection while the latter tests the effectiveness of CFI-LB's backward-edge protection (i.e., SafeStack [44]). We modified the existing PoC exploits [37], [42] to account for the difference in the code generated by GCC and Clang. These PoC exploits assume that ASLR is disabled. In reality, attackers often leverage information leaks to learn the necessary locations before the attack. We completely instrumented all the vulnerable programs with CFI-LB and use their test-suites for the CFG generation.

RIPE: RIPE is a 32-bit buffer-overflow benchmark suite. Since our prototype is based on the x86-64 architecture, we modified a few lines of the assembly code in the benchmark to make it work. During the evaluation, we first generated the dynamic CFG with valid inputs that did not trigger the buffer overflows. We then run the benchmark with CFI-LB and triggered the buffer overflow to hijack the control flow. All these attempts were detected and blocked by our system.

```
static PyObject *
sock_recvfrom_into(PySocketSockObject *s, PyObject *args, PyObject* kwds)
{
    ...
    if (recvlen == 0) {
        recvlen = buflen;
    }
    // there must be a check for overflow
+   if (buflen < recvlen) {
+       PyBuffer_Release(&pbuf);
+       PyErr_SetString(PyExc_ValueError,
+                       "buffer too small for requested bytes");
+       return NULL;
+    }

     readlen = sock_recvfrom_guts(s, buf.buf, recvlen, flags, &addr);
     ...
}
```

Fig. 7. CVE-2014-1912: the vulnerability and the patch

```
import socket
r, w = socket.socketpair()
w.send(b'\x90' * 305 + '\xc0' + '\x65' + '\x51' + '\x00')
r.recvfrom_into(bytearray(), 309)
```

Fig. 8. PoC exploit for Python CVE-2014-1912

```
000000000047caf0 <PyObject_Hash>:
  ...
  47cb23:  48 89 c1          mov    rcx,rax
  47cb26:  48 89 7d e0       mov    QWORD PTR [rbp-0x20],rdi
  47cb2a:  48 89 cf          mov    rdi,rcx
  47cb2d:  48 89 45 d8       mov    QWORD PTR [rbp-0x28],rax
  47cb31:  e8 ca 92 fa ff    call   425e00 <i_cfilb3_reference_monitor>
  47cb36:  48 8b 7d e0       mov    rdi,QWORD PTR [rbp-0x20]
  47cb3a:  48 8b 45 d8       mov    rax,QWORD PTR [rbp-0x28]
  47cb3e:  ff d0             call   rax
  ...
```

```
    if (tp->tp_hash != NULL)
        return (*tp->tp_hash)(v);
```

Fig. 9. The execution of the hijacked function pointer

CVE-2014-1912: this is a heap overflow in Python-2.7.6 [10]. The root cause is the missing check of the buffer size and the receive size in Python's socket module, as shown in Fig. 7. This vulnerability can be triggered by a malicious Python program. In Python, the script memory is allocated on the heap, sometimes adjacent to a Python object, which contains a number of function pointers that can be hijacked by exploiting this CVE. The PoC exploit in Fig. 8 overwrites the tp_hash function pointer. This function pointer can be executed by the indirect call (0x47cb3e) in Fig. 9. We took a close look at the

CFG for Python. This indirect call is protected by CFI-LB with three call-site sensitivity. It contains 329 ECs with only five valid targets (i.e., Python's internal hash functions). CFI-LB can detect any invalid target out of these five; and it can further distinguish these five valid targets by the contexts. For example, int_hash can only be called in 12 contexts.

```
- --- ftp/main.c:slurpstring() ---

406: char *sb = stringbase;     <--- This is our input. (can be massive)
407: char *ap = argbase;        <--- This buffer is 200 bytes.

458: S1:

463: case '\0':
464:  goto OUT;

474: default:
475:  *ap++ = *sb++;        <--- Heap overflow
476:  got_one = 1;
477:  goto S1;
478: }
- --------------------------------
backtrace at overflow:
main()->cmdscanner()->cd()->another()->makeargv()->slurpstring()
```

Fig. 10. Vulnerable code for EDB-ID 15705

```
./ftp
ftp> open
(to) b'\x90' * 745 + '\xd4' + '\x80' + '\x48' + '\x00'
usage: open host-name [port]
ftp> open
[iCFILB-LEVEL 1] Violation at 4796cb target to 4880d4 with context 46c804
```

Fig. 11. Simplified PoC for EDB-ID 15705 and the error message thrown by CFI-LB

EDB-ID 15705: this is a heap overflow in the FTP client of the GNU InetUtils package [19], as shown in Fig. 10. The vulnerability can be exploited through a malicious FTP command. Our PoC exploit uses a malicious open command to overwrite a function pointer that can later be called by the indirect call in cmdscanner function. CFI-LB uses one call-site sensitivity for this indirect call. The indirect call can only target two functions, setpeer and help. CFI-LB can detect any invalid function addresses and impose the more restrictive contexts on these two valid targets.

CVE-2016-2233: this is a stack-based overflow in HexChat-2.10.0. It can be exploited by a remote IRC server. Our PoC exploit leverages the exploits to overwrite the return address on the stack. When the stack canary is disabled, the program will cause a segmentation fault due to invalid memory access by function __gconv, instead of executing the malicious code. This is expected because SafeStack protects return addresses by segregating them into a separate safe stack. As such, the exploit failed to overwritten the return address, but instead overwrote other data on the stack.

Summary: the quantitative analysis with SPEC CPU 2006 and a complex program (NGINX) demonstrates the improvement in the security of CFI made by the adaptive call-site sensitivity. The experiment with the RIPE benchmark and real-world vulnerabilities shows that CFI-LB can block control flow hijacking attacks, hence maintaining control-flow integrity.

B. *Effectiveness of Localized Concolic Execution*

CFI-LB features the localized concolic execution to extend the (incomplete) dynamic CFG with as few false positives as possible. To evaluate this, we show the statistics to answer the following two questions: how close the dynamic and concolic CFGs combined is to the static CFG, and what are the main differences between the dynamic and concolic CFGs. The former can be answered with the set operation static-CFG $\setminus$ (dyn-CFG $\cup$ con-CFG) [4]; the latter can be answered with the set operations dyn-CFG $\setminus$ con-CFG and con-CFG $\setminus$dyn-CFG. We created the static CFG from CFL-AA, an Andersen-style alias analysis in the official LLVM source code. Because CFL-AA is context-insensitive, the static CFG accordingly has no contexts. As such, we removed all the contexts in the dynamic and concolic CFGs when calculating static-CFG $\setminus$ (dyn-CFG $\cup$ con-CFG). The other two set operations are calculated with two call-site sensitivity. Table V shows the results.

We found that the union of dyn-CFG and con-CFG, after removing the contexts, is rather close to the static-CFG, except for h264ref (see the 5th column). In other words, most of the run-time control flows can be directly verified by CFI-LB without the extra offline verification. Meanwhile, our localized concolic execution performs well in finding new control transfers that do not exist in dyn-CFG (the 6th and 7th columns). For example, con-CFG has 308 more entries than dyn-CFG and only misses 14 entries from it for 403.gcc.

To check the correctness of con-CFG, we randomly picked some parts of it and manually analyzed them. Overall, we did not find any false positives for these benchmarks (false positives are possible as stated earlier.) Using sjeng as an example, we found that it contains a function pointer dispatch table, which consists of six function pointers to handle the normal cases and another one to handle errors. Because the inputs to generate dyn-CFG only contain valid data, dyn-CFG can never include this target. By using the concolic execution, we can explore all the paths of the related functions and successfully discovered this target. We also found some false negatives caused by the limitations of the concolic execution in general and the specific tool we use. For example, we found the SMT solver failed to generate the strings starting with the '#' character, which would otherwise lead to three new targets.

To further test the effectiveness of this technique, we conducted the experiments to compare con-CFG generated from the *small* inputs to dyn-CFG generated from the *large* inputs. Specifically, SPEC CPU2006 includes both a smaller test data set and a much larger reference data set. We run our localized concolic execution based on the execution history generated from the test data set (called con-CFG-t), and compared con-CFG-t to the dynamic CFG generated from the reference data set (called dyn-CFG-r). The results are shown in Table VI. It demonstrates that our concolic execution can discover a significantly larger CFG from the small inputs, comparable to the CFG generated from the much larger inputs. For example, if we run 403.gcc on the *reference* data set but with the

[4] $\setminus$ returns the elements in the first operand but not in the second operand.

TABLE V

Comparing static, dynamic, and concolic CFGs for the Spec CPU 2006 benchmarks. Column 2 to 4 show the total number of entries in these CFGs, respectively. Note that number of static-CFG is not directly comparable to these of dyn-CFG and con-CFG because the latter two CFGs have contexts (hence more entries).

Benchmark	static-CFG	dyn-CFG	con-CFG	static-CFG $\setminus$(dyn-CFG' $\cup$ con-CFG')	dyn-CFG $\setminus$ con-CFG	con-CFG $\setminus$ dyn-CFG
400.perlbench	879	1374	1387	41 (4.66%)	0 (0%)	13 (0.94%)
401.bzip2	20	12	16	4 (20%)	0 (0%)	4 (25%)
403.gcc	2198	3831	4125	94 (4.28%)	14 (0.37%)	308 (7.47%)
445.gobmk	957	1882	1971	79 (8.25%)	23 (1.22%)	112 (5.68%)
456.hmmer	52	47	59	6 (11.54%)	0 (0%)	12 (20.34%)
458.sjeng	7	6	7	0 (0%)	0 (0%)	1 (14.29%)
464.h2564ref	711	262	479	206 (28.97%)	12 (4.58%)	229 (47.81%)

TABLE VI

Comparing concolic and dynamic CFGs. dyn/con-CFG-t is derived from the small test inputs; dyn/con-CFG-r is derived from the large reference inputs. Our localized concolic execution can discover most of the control transfers in the dyn-CFG-r using only the small inputs.

Benchmark	dyn-CFG-r	dyn-CFG-t	con-CFG-t	dyn-CFG-r $\setminus$ dyn-CFG-t	dyn-CFG-r $\setminus$ con-CFG-t	Discovered
400.perlbench	1374	449	1051	925	323 (23.51%)	602
401.bzip2	12	12	16	0	0 (0%)	0
403.gcc	3831	2196	3929	1635	53 (1.38%)	1582
445.gobmk	1882	1102	1833	780	49 (2.60%)	731
456.hmmer	47	3	58	44	1 (2.13%)	43
458.sjeng	6	6	7	0	0 (0%)	0
464.h264ref	262	240	473	22	18 (6.87%)	4

CFG generated from the *test* data set, only 53 (1.38% of the total control transfers in dyn-CFG-r, the 6^{th} column) context-sensitive control transfers need to be validated by the static CFG. Note that the dynamic CFG from the reference data set has $1,635$ (the 5^{th} column) more entries than that from the test data set; most of which are successfully discovered by our localized concolic execution.

Summary: this evaluation demonstrated the effectiveness and efficiency of our proposed localized concolic execution in extending the dynamic CFG close to the static CFG, even with a small input data set. Consequently, our multi-scope CFG can efficiently verify most run-time control transfers online.

C. Performance Evaluation

We evaluated the overhead of CFI-LB on the Intel core-i7 6700 processor (Skylake) with a base frequency of 3.4GHz and 16GB of memory, running the 64-bit Ubuntu 16.04.3 LTS system. We used the SPEC CPU2006 benchmarks, including all the SPECint 2006 benchmarks and all the C/C++ benchmarks in SPECfp 2006, and the NGINX benchmark for the evaluation. Note that a few benchmarks had no overhead because either their code did not use indirect calls or their indirect calls were not executed by the benchmark inputs. In the following, we exclude three such benchmarks from the average (Average_ex in Fig. 12). For C++ based benchmarks, we applied CFI-LB to both the C-style indirect calls and virtual calls. We tested the performance of CFI-LB by the following four configurations: CFI-LB without SafeStack or TSX, with SafeStack only, with TSX only, and with both features.

The forward edge protection of CFI-LB incurred about 2.7% of overhead on average, with a maximum of 5% (xalancbmk). Note that the performance overhead of CFI-LB is decided by how frequently indirect calls are executed. For example, even though sjeng only had one executed indirect call, that indirect call was executed $775,046,817$ times during the benchmark. Our prototype relies on the SafeStack to protect return addresses. SafeStack only incurred an addition 0.5% of overhead on average, with a maximum of about 1.5%. Interestingly, it had negative performance impact on some benchmarks. This is consistent with the original system [31].

We measured the performance of CFI-LB both with and without the TSX support. TSX-based hardware transactional memory is used by CFI-LB to prevent race conditions against its reference monitors. As shown in Fig. 12, the average performance overhead introduced by TSX was about 1.4%. We also measured the false failure rate of transactions caused by the false sharing or cache conflicts. The failure rate was low at about 0.002%. Therefore, the false transaction abort was not a concern for performance.

The average performance overhead of CFI-LB with both SafeStack and TSX was about 4.8%, with a maximum of 8.4%. Note that the overhead introduced by SafeStack and TSX cannot be simply added together. SafeStack changes the stack layout and the program behavior. This may subtly change the program's performance under TSX: TSX is enforced at the cache line level. When the stack layout is changed by SafeStack, the program may have a different cache profile that further affects the overhead of TSX. For example, it seems that adding the TSX support incurred no additional overhead for 403.gcc but more overhead for benchmarks such as NGINX.

Other than the run-time performance, our offline analysis process took about 4 hours for each benchmark measured on an Intel Xeon E5-2630v2 (2.60GHz) machine with 32GB memory. We consider this performance reasonable since the

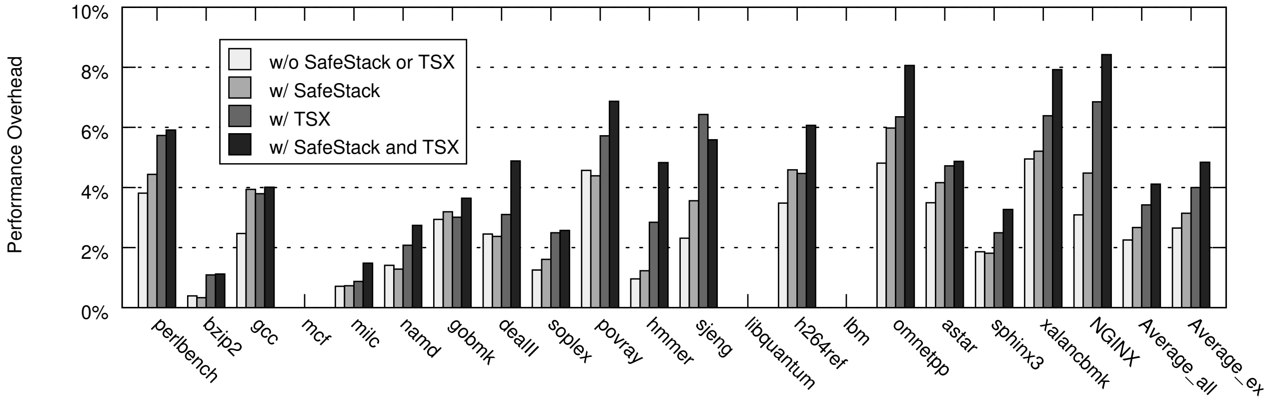

Fig. 12. Performance overhead, Average_ex shows the average overhead excluding the three benchmarks that have no overhead.

offline analysis is conducted only once offline.

Summary: on average, CFI-LB introduced a low performance overhead: 2.7% for the forward-edge protection and 4.8% for the full protection.

V. Discussion

In this section, we discuss the potential improvements and the future work for CFI-LB.

First, CFI-LB is explicitly designed to protect its reference monitors from race conditions. Some CFI systems are implicitly protected from race conditions because their reference monitors are encoded in the registers only [2]. Any CFI systems that rely on the compiler for register allocation cannot provide this guarantee. This includes most recent CFI systems, which implement their instrumentation in the LLVM IR [1], [4], [34]. Only a few CFI systems are implemented by direct binary instrumentation [2], [12]. Even such systems require careful vetting to ensure that they are not susceptible to race conditions. Moreover, this approach does not work for context-sensitive CFI systems because of x86's limited number of registers. We address this challenge with TSX. Note that CFI-LB cannot use the TSX instructions inside the existing TSX-protected code sections of the program. Such conflict is unlikely because TSX is often used to protect short critical sections, which typically do not contain indirect calls.

Second, the security of CFI is decided by both the average and the largest EC sizes. Unfortunately, our evaluation shows that call-site sensitivity may not be effective in reducing the largest EC size. The ability to reduce the largest EC size is decided by the program structure itself and the maximum level of call-site sensitivity. For example, the largest indirect call in `403.gcc` is called by a recursive function. This makes it difficult, if not impossible, for call-site sensitivity to reduce the largest EC size. Path sensitivity is not very effective either. For example, the largest EC size for SPEC CPU2006 benchmarks is 218 for PittyPat [18]. A follow-up work of PittyPat significantly improves this situation [27]. In addition, there are cases where increasing sensitivity helps but could lead to the explosion of the number of ECs. To address that, we could use a more powerful model of the call stack, e.g., a regular expression. This will allow us to support recursive callers by combining the consecutive instances of the same caller. We can also identify common programming patterns that lead to this ineffectiveness and automatically transform the program to make it more amicable to call-site sensitivity.

Third, the static CFG is constructed with the points-to analysis, a known NP-hard problem [55]. Precise points-to analysis algorithms often rely on context sensitivity to improve precision; but they do not scale well for C/C++ programs. Many published precise points-to algorithms were evaluated only with toy programs. The public availability of these algorithms are even less common, let alone well-maintained. After a long search, we find the best available context-sensitive points-to algorithm at the time of writing is the DSA algorithm, which has not been maintained for a long time and contains known algorithmic errors [32], [33]. To temporarily relieve this situation, we propose the multi-scope CFG and its localized concolic execution to significantly extend the dynamic CFG. Our system can easily adopt a static context-sensitive CFG when it becomes available.

Fourth, CFI-LB relies on offline analysis to check control transfers within the static CFG but outside the dynamic CFGs. An identified benign control transfer will be added to the dynamic CFG, while a malicious one can be added to a blacklist. This will gradually increase the scope of the dynamic CFGs, making the offline analysis less frequent. The offline analysis should be (mostly) automated to be useful. A simple strategy is to black-list every abnormal control transfer that leads to the program crash. Given the many exploit mitigation mechanisms deploy in application, this strategy is potentially effective. Overall, this is a complex problem that deserves its own line of research [9]. We leave it as a future work. We would like to mention that our localized concolic execution makes the need for offline analysis much less burdensome – as shown in Table VI, most benchmarks require analysis of less than 55 control transfers.

Lastly, our localized concolic execution tries to explore multiple paths that immediately lead to an indirect call in

TABLE VII

Compare some CFI systems that use hardware support

CFI Systems	CCFI	PathArmor	PittyPat	CFI-LB
Context	context insensitive	path sensitive	path sensitive	call-site sensitive
Hardware support	Intel AES-NI	Intel last branch record	Intel processor tracing	software-based, Intel TSX/CET if available
Coverage	authenticate all code pointers	limited to paths before seven sensitive syscalls	entire execution leading to the sensitive syscalls	every indirect call/jump, relying on the shadow stack to protect returns
CFG	type-based CFG	on-demand, constraint-driven context-sensitive CFG	no pre-computed CFG, using online validation	multi-scope CFGs
Kernel changes	`sigaction` modifications to verify code pointers	kernel module to monitor path and intercept syscalls	kernel module to control Intel PT and intercept syscalls	no kernel changes

order to discover new targets. A complimentary strategy is to follow the def-use chain backwards and select the function that *defines* the related function pointer as the starting function. The intuition is that programmers often conditionally assign to the function pointer in a single function. We can explore all the paths of this function so that the function pointer can be assigned to other values. However, this implies that the target indirect call is included in the captured execution history. Our current approach does not have this constraint. In addition, the “define” function could be too far from the “use” function for the concolic execution to handle. We plan to combine both strategies to further improve the concolic CFG.

VI. Related Work

In this section, we discuss the work closely related to CFI-LB. In their seminal work, Abadi et al. introduced the key concept of Control Flow Integrity (CFI) [2] that has inspired a long stream of research [1], [6], [8], [14], [15], [16], [18], [23], [36], [38], [39], [41], [50], [52], [53], [54]. The original implementation of CFI uses a tag-based enforcement mechanism. As such, it suffers from the imprecision caused by equivalence classes, which is a common limitation of the context-insensitive CFI systems. A context-insensitive CFI that does not have this problem is HyperSafe [50], which uses a dedicated jump table for each indirect call. However, the main purpose of HyperSafe is to enforce the CFI for a hypervisor. Accordingly, its performance overhead was not evaluated with the standard benchmarks, such as SPEC CPU2006. A recent survey by Burow et al. provides a comprehensive comparison of the context-insensitive CFI systems [3]. Different from these systems, CFI-LB is a context-sensitive system.

Recently, Intel has introduced numerous security features in their processors. Many of these features are used in recent CFI systems [4], [13], [18], [22], [24], [34], [48]. Table VII compares some of these CFI systems. For example, CCFI (Cryptographic CFI) leverages the hardware AES acceleration to cryptographically authenticate code pointers in order to protect them from malicious modification [34]. CCFI is context-insensitive and uses a type-based CFG, in which an indirect call can transfer to any address-taken functions that have a compatible prototype. CFI systems that rely on the hardware support often require to change the kernel, for example, to control the hardware feature. Although CCFI does not need the kernel privilege to access AES-NI, it still has to change the signal handling code to authenticate the user signal handler. Because CCFI needs to cryptographically authenticate every code pointer, its performance overhead is rather high. The recently announced pointer authentication on the ARM v8.3 platform can accelerate the pointer authentication in hardware [5]. However, there is no publicly available SoC that implements this feature yet. CFIXX proposes the object-type integrity to protect virtual calls in C++ programs [4]. Specifically, it stores the mapping between the object and its type in the meta-data, and protects the meta-data with the Intel MPX technology. CFIXX can prevent a wide variety of `vtable` hijacking attacks [45]. CFI-LB can prevent most `vtable` hijacking attacks because of its precise CFI policy. Object-type integrity is a complementary policy to CFI [4].

PathArmor [48] and PittyPat [18] are two closely related systems. They both implement the path-sensitivity CFI policy. PathArmor relies on the Intel last branch record (LBR) to record the most recent 16/32 branches. It then employs an on-demand constraint-driven method to calculate a small relevant part of the context-sensitive CFG and further validates the control flow. For performance reasons, PathArmor only validates the immediate paths before a small selected set of sensitive syscalls. As such, PathArmor can only provide a partial protection to the process. Meanwhile, PittyPat uses the more powerful Intel processor tracing (PT) that can continuously track a process’ control flow. Intel PT hence has higher performance and storage overhead than LBR. To address that, PittyPat redirects the process tracing data to a different process and relies on another CPU core to offload the verification. The protected process and the verifier are synchronized at the selected syscalls, i.e., the verification is only performed at these (ten) syscalls. However, since Intel PT provides a more complete history, PittyPat can verify the whole execution path leading to the syscall. As such, PittyPat has broader coverage than PathArmor, but it reduces the usable CPU cores and limits the number of processes it can protect simultaneously. CFI-LB instead enforces the call-site sensitivity. Technically, path sensitivity is more fine-grained than call-site sensitivity because they can take individual branches into consideration.

CFI-LB excels at enforcing the protection for the whole program all the time. It also has much lower overhead, given that both PathArmor and PittyPat only enforces the protection at the selected syscall boundary.

The CFG construction is still a mostly unsolved problem for CFI systems. Many CFI systems use a coarse-grained CFG. For example, some CFI systems assume that each indirect call can legitimately transfer to any address-taken functions [20], [25], [26], [47]. An improvement over that is to only allow an indirect call to transfer to address-taken functions that have compatible types [1], [34], [38]. To support multiple modules, modular CFI allows run-time updates to the CFG in order to protect the inter-module indirect calls [38]. The way PittyPat verifies the control flow is interesting. It does not maintain a CFG; it instead uses the recorded execution path to calculate the valid run-time control transfers. This is somewhat similar to PathArmor's constraint-based CFG construction in that both use the past execution history to constrain the possible valid control transfers. CFI-LB instead uses the multi-scope CFG that combines a context-insensitive CFG with context-sensitive dynamic and concolic CFGs. Our experiments show that our localized concolic execution can significantly extend the dynamic CFG with few false positives. There are also many systems that do not rely on the CFG, but use heuristics to detect anomaly in the control flow. For example, kBouncer and ROPecker [7], [40], look for anomalous control patterns at sensitive locations.

VII. Conclusion

We have presented the design, implementation, and evaluation of CFI-LB, an adaptive call-site sensitive CFI system. CFI-LB has two unique features: adaptive call-site sensitivity and the multi-scope CFG. The former balances the security and the performance by allowing each indirect call to decide its own level of sensitivity; the latter aims at improving the security of CFI even if a precise context-sensitive CFG is not available by using multiple CFGs and combining the online and offline verification. In addition, CFI-LB is the first CFI system that can explicitly guarantee the atomicity of its reference monitors. Our evaluation shows that CFI-LB can significantly improve the security over the traditional context-insensitive CFI systems and incur a small, acceptable performance overhead.

VIII. Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. Zhi Wang was partially supported by National Science Foundation (NSF) under Grant 1453020; Yajin Zhou was partially supported by the National Natural Science Foundation of China (NSFC) under Grant 61872438. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or NSFC.

References

[1] Niu, Ben and Tan, Gang , "Per-input Control-flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 914–926.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[3] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 16:1–16:33, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3054924

[4] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CFIXX: Object Type Integrity for C++," in *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018.

[5] A. Can, A. Krishnaswamy, and R. Turner, "Code Pointer Authentication for Hardware Flow Control," Dec. 6 2016, uS Patent 9,514,305.

[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the 24th USENIX Security Symposium*, vol. 14, 2015, pp. 28–38.

[7] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG *et al.*, "ROPecker: A Generic and Practical Approach for Defending against ROP Attack," 2014.

[8] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-flow Integrity for Commodity Operating System Kernels," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 292–307.

[9] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, "Retracer: Triaging crashes by reverse execution from partial memory dumps," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 820–831. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884844

[10] "CVE-2014-1912," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1912.

[11] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15, 2015.

[12] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A Framework to Mitigate Control-flow Attacks on Smartphones," in *NDSS*, vol. 26, 2012, pp. 27–40.

[13] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-assisted Flow Integrity Extension," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 74.

[14] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted Fine-grained Control-flow Integrity: Towards Efficient Protection of Embedded Systems against Software Exploitation," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[15] L. Davi and A.-R. Sadeghi, "Building Control-flow Integrity Defenses," in *Building Secure Defenses Against Code-Reuse Attacks*. Springer, 2015, pp. 27–54.

[16] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection," in *Proceedings of the 23Rd USENIX Conference on Security*, ser. SEC'14, 2014.

[17] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[18] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient Protection of Path-sensitive Control Security," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 131–148. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding

[19] "EDB-ID-15705," https://www.exploit-db.com/exploits/15705/.

[20] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software Guards for System Address Spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 75–88.

[21] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-grained Control-flow Integrity," in *Proceedings of the 22nd ACM*

SIGSAC Conference on Computer and Communications Security. ACM, 2015, pp. 901–913.

[22] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding Control Flows Using Intel Processor Trace," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 585–598.

[23] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-flow Integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14, 2014.

[24] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent Backward-edge Control Flow Violation Detection Using Intel Processor Trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 173–184.

[25] B. Hardekopf and C. Lin, "Semi-Sparse Flow-sensitive Pointer Analysis," in *Proceedings of the 2009 ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 2009.

[26] M. Hind and A. Pioli, "Which Pointer Analysis should I Use?" in *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 5. ACM, 2000, pp. 113–123.

[27] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1470–1486. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243797

[28] *Intel 64 and IA-32 Architectures Software Developerś Manual*, Intel.

[29] Intel, "Control-flow Enforcement," https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2018.

[30] Intel, "Intel Pin Tool," http://intel.ly/2jc3TSy, 2018.

[31] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer Integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 147–163. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[32] C. Lattner, A. Lenharth, and V. Adve, "Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, 2007.

[33] LLVM Forum, "LLVM DSA - Reproduce the Result in PLDI 07 Paper," http://lists.llvm.org/pipermail/llvm-dev/2015-May/085359.html, 2018.

[34] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically Enforced Control-flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 941–951.

[35] Microsoft, "The Evolution of CFI Attacks and Defenses," https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2018_02_OffensiveCon, 2018.

[36] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque Control-flow Integrity," in *Proceedings of the 22th Network and Distributed System Security Symposium*, ser. NDSS '15, 2015.

[37] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the Reproducibility of Crowd-reported Security Vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/mu

[44] "Clang SafeStack," https://clang.llvm.org/docs/SafeStack.html.

[38] B. Niu and G. Tan, "Modular Control-flow Integrity," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 577–587, 2014.

[39] Niu, Ben and Tan, Gang, "RockJIT: Securing Just-in-time Compilation Using Modular Control-flow Integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1317–1328.

[40] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing," in *USENIX Security Symposium*, 2013, pp. 447–462.

[41] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained Control-flow Integrity through Binary Hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[42] "PoC for CVE's," https://github.com/VulnReproduction/LinuxFlaw.

[43] Quarkslab, "Triton Symbolic Engine," http://bit.ly/2AKOLCX, 2018.

[45] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*. IEEE, 2015.

[46] N. Stojanovski, M. Gusev, D. Gligoroski, and S. J. Knapskog, "Bypassing Data Execution Prevention on Microsoft Windows xp sp2," in *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*. IEEE, 2007, pp. 1222–1226.

[47] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-edge Control-flow Integrity in GCC & LLVM," in *USENIX Security Symposium*, 2014, pp. 941–955.

[48] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-sensitive CFI," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.

[49] C. Vulnerabilities and Exposures, "CVE List of Memory Corruption," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=memory+corruption.

[50] Z. Wang and X. Jiang, "Hypersafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-flow Integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 380–395.

[51] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime Intrusion Prevention Evaluator," in *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM, 2011.

[52] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting Violation of Control-flow Integrity Using Performance Counters," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12.

[53] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13, 2013.

[54] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13, 2013.

[55] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the Value of Static Analysis for Fault Detection in Software," *IEEE transactions on software engineering*, vol. 32, no. 4, pp. 240–253, 2006.