Resilient CFI: Compiler-based Attack Origin Tracking with Dynamic Taint Analysis

Oliver Braunsdorf[0000−0001−9275−9978], Stefan Sessinghaus, and Julian Horsch[0000−0001−9018−7048]

Fraunhofer AISEC, Lichtenbergstr. 11, 85748 Garching near Munich, Germany
<firstname>.<lastname>@aisec.fraunhofer.de
https://www.aisec.fraunhofer.de/

Abstract. Over the last decade, many exploit mitigations based on Control Flow Integrity (CFI) have been developed to secure programs from being hijacked by attackers. However, most of them only abort the protected application after attack detection, producing no further information for attack analysis. Solely restarting the application leaves it open for repeated attack attempts. We propose *Resilient CFI*, a compiler-based CFI approach that utilizes dynamic taint analysis to detect code pointer overwrites and trace attacks back to their origin. Gained insights can be used to identify attackers and exclude them from further communication with the application. We implemented our approach as extension to LLVM's Dataflow Sanitizer, an actively maintained data-flow tracking engine. Our results show that control-flow hijacking attempts can be reliably detected. Compared to previous approaches based on Dynamic Binary Instrumentation, our compiler-based static instrumentation introduces less run-time overhead: on average 3.52x for SPEC CPU2017 benchmarks and 1.56x for the real-world web server NginX.

Keywords: Software Security · Control Flow Integrity · Resiliency · Taint Analysis · Dynamic Information Flow Tracking · LLVM

1 Introduction

Control-flow hijacking attacks are one of the most harmful threats for today's software landscape. They enable attackers to execute malicious code inside a victim program to gain complete control over its execution context and eventually further infiltrate whole IT systems. Depending on purpose and leverage of IT systems, attackers could misuse them to extract confidential information, manipulate sensitive data or control physical operations.

Because of the severe consequences, motivation for academia and industry is high to create new approaches to effectively mitigate control-flow hijacking attacks of programs written in low-level programming languages like C and C++. Developed countermeasures like Data Execution Prevention (DEP), Stack Canaries and Address Space Layout Randomization (ASLR) proved to be practical and are now used by default in many runtime environments. They

successfully protect against code injection attacks but are only partially effective against advanced exploit techniques such as *return-to-libc* or *Return-Oriented Programming (ROP)*. Therefore, within the past 15 years, academia developed a countermeasure called *Control Flow Integrity (CFI)*. Abadi et al. originally introduced CFI [1] as a technique which statically instruments binaries to secure *forward-edge* (indirect jumps, indirect function calls) and *backward-edge* (function returns) control-flow instructions with additional run-time checks. Within the last years, this technique has been taken up by many publications which improved concepts for backward-edge CFI and forward-edge CFI. As CFI has proven an efficient and mostly effective countermeasure, it nowadays becomes more and more adopted in industry standard compilers [16].

Unfortunately, most (if not all) existing CFI approaches only consider detection of control-flow hijacking attacks. Their default behavior upon attack detection is to abort the program to prevent further exploitation. While "Detect and Abort" might be a reasonable strategy to prevent extraction of confidential information or manipulation of sensitive data, it can result in loss or damage of data and affects the availability of provided services. Service providers are well prepared for occasional restart and recovery procedures after program abort but in case of a targeted attack, the attacker can start the exploit over and over again. If no appropriate countermeasures are taken between aborting and restarting, subsequent exploitation attempts can cause non-negligible downtimes. For providers of critical services, this can be serious threat to availability.

In this paper, we present *Resilient CFI (ReCFI)*, an approach for control-flow integrity which enables advanced attack reaction strategies to counter repeated control-flow hijacking attempts via similar attack vectors. Resilient CFI (ReCFI) is able to determine the source of attacks at program level and extract information about attackers which can be used to exclude them from further communication with the program. ReCFI is designed as a compiler plug-in, providing both forward- and backward-edge CFI. It aims for small run-time overhead as well as zero false negatives and false positives in order to be a solution for attack mitigation and not only being used as a tool for testing or debugging. As a basis for our work, we employ *dynamic taint analysis* to detect malicious overwrites of code pointers, track back the attack to its originating interface and generate useful information for identifying the attacker.

The idea of utilizing dynamic taint analysis for detecting code pointer overwrites is not completely new. In the past, approaches have been proposed to use specialized hardware in order to track data-flow and check the integrity of code pointers efficiently while the program is executed [15,5,10,8]. Unfortunately, specialized hardware is too expensive to deploy those solutions widely. Newsome and Song [14], followed by other authors [6,7,11], proposed software-implemented taint-tracking engines based on Dynamic Binary Instrumentation (DBI) to ensure control-flow integrity on commodity systems. However, dynamic instrumentation causes unacceptable run-time delays, hindering practical usability of those approaches.

With ReCFI, we propose a compiler-based approach that statically inserts additional instructions to a program at compile-time, thus eliminating the runtime overhead for dynamic instrumentation. Our contributions are summarized in the following:

- We provide a concept for utilizing dynamic taint analysis to detect malicious overwrites of code pointers which additionally provides the ability to generate information to identify the attack origin.
- We implement[1] our concept as an extension to LLVM's *Dataflow Sanitizer*, an actively maintained data-flow tracking framework.
- We implement security measures necessary for static instrumentation solutions to protect against attackers modeled with *contiguous write* capabilities.
- We evaluate the performance using SPEC CPU2017 benchmarks and conduct a case study with the popular NginX web server to exemplify how ReCFI can be applied to complex real-world programs and protect them from repeated exploitation attempts.

We present our design goals and the attacker model under which we developed ReCFI in Section 2. We explain the concept of ReCFI in Section 3 before describing details of its implementation in Section 4. In Section 5, we show how ReCFI can be applied to the NginX web server as a case study. We discuss the security properties of ReCFI in Section 6 and evaluate its performance in Section 7. In Section 8, we summarize related work before concluding the paper in Section 9.

2 Design Goals and Attacker Model

ReCFI aims to prevent repeated attacks on the control-flow integrity of a program. It is a security mechanism designed to (1) detect manipulation of code pointers resulting from memory corruption-based attacks like heap or stack-based buffer-overflows, (2) identify the origin of an attack, i.e., the interface used by the attacker to inject malicious input, (3) provide detailed information about the origin of an attack gained directly from the running program, and (4) support programs which need to run natively without an interpreter or virtual machine environment.

We assume that the program protected by ReCFI contains memory-corruption errors. An attacker is able to (remotely) send malicious input data to exploit those errors and thereby gain the following capabilities:

Arbitrary read: The attacker can read from arbitrary memory locations of the target program's address space.

Contiguous write: Starting from the memory location of a vulnerable buffer or array variable, the attacker can overwrite adjacent memory locations (that are not write-protected by the operating system).

[1] ReCFI source code published here: https://github.com/Fraunhofer-AISEC/ReCFI

This is a realistic attacker model which, in practice, means we assume that all attacks utilize buffer over- and underflows on the stack or heap. ReCFI is implemented to also detect more powerful attackers which are able to overwrite code pointers *not* adjacent to the vulnerable buffer but at arbitrary locations. However, we rely on the integrity of taint tracking metadata as ground truth for our attack detection algorithm and therefore cannot assume an attacker with *arbitrary write* capabilities because that would include the ability to maliciously overwrite the metadata.

3 Resilient CFI

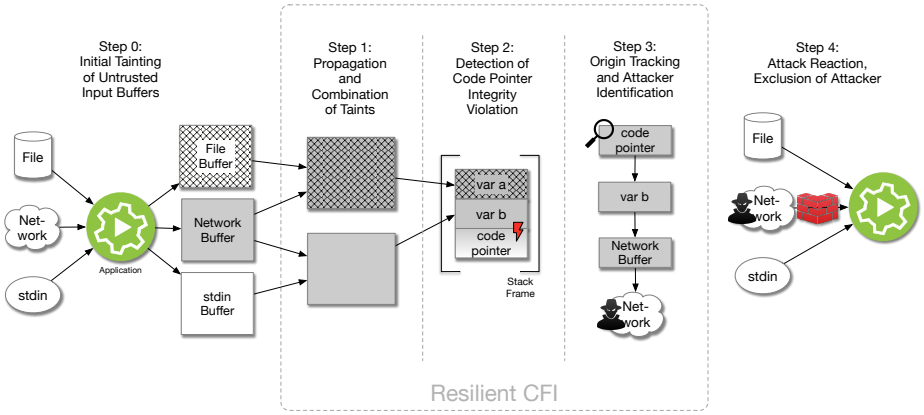

Fig. 1: Overview of ReCFI's steps to protect an application with input from standard input stream (trusted), and network and filesystem (untrusted).

ReCFI is a compiler-based approach for Control Flow Integrity (CFI), driven by dynamic taint analysis. It protects both, forward and backward control-flow edges and provides information about the origin of the attack. ReCFI works by assigning taint labels to untrusted program input data, propagating these taint labels in memory throughout program execution and verifying the absence of taint of code pointers before execution of control-flow instructions. We designed ReCFI as a compiler-based security mechanism and implemented it on top of the LLVM[2] compiler infrastructure. At compile-time, ReCFI instruments the program at LLVM IR level to insert additional instructions. At run-time, original and instrumented instructions are executed together natively to perform taint propagation simultaneously and to detect attacks as they happen. The LLVM developer community maintains a framework for dynamic taint analysis called *Dataflow Sanitizer (DFSan)*[3], which we extended to implement our CFI approach.

[2] `https://llvm.org/`
[3] `https://clang.llvm.org/docs/DataFlowSanitizer.html`

Figure 1 illustrates ReCFI's methodology, which we divided into sequential steps that are described in the following.

Step 0: Initial Input Tainting. ReCFI requires an initial tainting of untrusted program input. For most applications, network or file I/O is untrusted. Depending on the application domain, other buffer variables containing program input from, e.g., command-line arguments, environment variables, or memory-mapped peripherals could also be untrusted. In the example shown in Figure 1, file and network interfaces are untrusted and their respected buffers get tainted while the standard input stream is considered trustworthy. Finding the complete and sound set of untrusted input buffers for a given program is a highly application-dependent task. Our approach does not aim to provide a general solution to this task. Instead, ReCFI provides a simple API that can be used by application developers to set the initial taint label of variables in the program's source code.

Step 1: Taint Propagation. To model the program's taint state at run-time, we utilize DFSan to split up the program's memory space and set up a shadow memory area as shown in Figure 2. For every byte of application memory, the shadow memory contains the corresponding taint label. To keep track of the program's taint state, DFSan instruments every data-flow instruction with additional instructions to propagate taint labels. Before an instruction is executed, the taint labels of every instruction operand are loaded from shadow memory and propagated according to the semantics of the instruction. For binary operations, e.g., an arithmetic instructions, DFSan combines the taint of both operands, generating a new *union taint label*. As an example, in Figure 1, the taints of the file and network buffers are combined, e.g., because of an XOR operation within a cryptographic function decrypting a network packet with a key read from a file. Hence a new *union taint label* is created and assigned to the decryption result buffer. DFSan provides the ability to regain the original taint labels from the union taint label they have been combined to. This is essential to ReCFI's *Attack Origin Tracking* step.

Step 2: Attack Detection. In addition to DFSan's instrumentation of data-flow instructions, ReCFI also instruments control-flow instructions. Our approach provides both, forward- and backward-edge control-flow integrity by instrumenting every indirect jump, call and return instruction. In front of any of those instructions, ReCFI inserts additional instructions to check the taint label of the target address. If the taint label indicates that the target address is tainted, then the current program execution is considered to be under attack because a tainted target address implies that the control flow is directly controlled by input from untrusted sources. In the exemplary program execution illustrated in Step 2 of in Figure 1, a code pointer (e.g. the return address) is overwritten with data from variable b as part of an attack that exploits a buffer-overflow vulnerability. In this case, ReCFI detects the tainted return address and prevents the program from

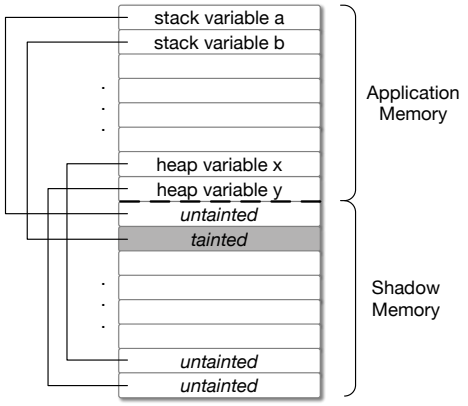

Fig. 2: Memory layout of an application protected by ReCFI.

executing the control flow instruction with the corrupted code pointer. Instead invokes an attack reaction routine which acts as the entry point to Step 3.

Step 3: Attack Origin Tracking. After an attack has been detected, ReCFI analyzes the taint label of the corrupted code pointer. Thereby, it recursively resolves union taint labels to their original taint labels to eventually identify from which particular program input source the code pointer has been derived from. ReCFI supports 2^{16} different taint labels and therefore can distinguish between many different input sources, respectively different potential attackers. In the example in Figure 1, ReCFI is able to resolve the taint label of the corrupted code pointer and trace the attack back to an attacker who used the network interface. The quality of identification of a particular attacker strongly depends on the granularity of input tainting, as shown in the following three examples:

1. If the same taint label is assigned to every data packet read from the network interface, ReCFI is only able to identify that the attacker is coming from the network.
2. If a new taint label is generated per communication peer, then ReCFI can narrow down the attacker's identity, e.g., to an IP or MAC address, a username, or a TLS certificate—depending on how communication peers are distinguished when generating the new taint label in the first place.
3. If a new taint label is generated per incoming network packet, ReCFI can deduce the actual network packet that triggered the vulnerability. Hence, it is possible to conduct advanced analyses of the packet's payload to identify the attacker.

The more fine-grained the input tainting, the more precise an attacker can be identified, which can be highly valuable for Step 4.

Step 4: Attack Reaction. After an attack has been detected, an appropriate reaction has to be conducted to mitigate subsequent similar attacks. Those reactions are

highly application dependent. Nevertheless, we want to briefly point out two common patterns for attack reactions enabled by ReCFI in the following:

Attack reactions can utilize filtering mechanisms to blacklist communication with peers identified as attackers. In networked applications, a firewall can be configured based on the network address (IP, MAC, etc.) of a malicious packet which can be identified by ReCFI. Consequently, after restarting the application, the attacker will not be able to conduct the exploit again from the same network address. In applications authenticating their users using username-password combinations or X.509 certificates, ReCFI can be also used to create a new taint label for every user and deny access to the service by blacklisting usernames or revoking certificates.

Another possible attack reaction could be to *selectively* apply memory-safety techniques to the program under attack, e.g. adding bounds checks to overflown buffers or move them into guarded memory locations[4]. Provided with detailed information about the attack origin generated by ReCFI, those memory-safety techniques can be selectively applied only to data structures on the attack path, thus minimizing the performance and memory overheads.

Besides reactions that aim to protect a single program, with ReCFI it is also possible to forward attack information via network sockets or IPC mechanisms to a monitoring component in order to contribute in assessing the overall security status of a system and select appropriate actions to protect the system.

4 Implementation Details

The implementation of ReCFI is based on *Dataflow Sanitizer (DFSan)*, a framework for dynamic taint analysis which is actively maintained as part of LLVM. Since DFSan is not designed as a CFI mechanism, we had to implement additional security measures to protect programs according to our attacker model. In this section, we summarize relevant implementation internals of DFSan and describe the modifications we conducted in order to implement ReCFI.

Memory Layout and Taint Representation. DFSan divides the process' virtual memory into application memory and shadow memory, as schematically depicted in Figure 2. Every byte of application memory is associated with two bytes (16 bit) of shadow memory representing its corresponding taint label. The mapping between application memory and shadow memory is calculated via a bitmask.

With DFSan's taint labels being two bytes (16 bit) in size, ReCFI can differentiate between $2^{16} - 1$ potential attackers (excluding the "untainted" label). By default, DFSan aborts the application if more than 2^{16} taint labels are created. To support use cases where more taint labels are needed, we modified DFSan to wrap around instead of aborting, and reuse taint label identifiers. ReCFI emits a warning every time the number of taint labels wraps around. Security implications of this wrap-around strategy are further discussed in Section 6.

[4] As realized by Electric Fence (`https://linux.die.net/man/3/efence`) or LLVM's Address Sanitizer (`https://clang.llvm.org/docs/AddressSanitizer.html`)

Checking the Integrity of Code Pointers. For the implementation of Step 2 of ReCFI, we additionally instrument control-flow instructions. Implementing CFI checks for indirect jumps and function calls in LLVM is straight-forward because the target address pointer is an argument of the call/jump instruction. Therefore, ReCFI just has to calculate the shadow memory location corresponding to the pointer's location which is represented in LLVM IR.

For return instructions, the location of the return address is architecture-specific and therefore only known by the machine code generation backend of LLVM. On `x86` and `aarch64` processor architectures the LLVM intrinsic function `llvm.addressofreturnaddress`[5] can be utilized to obtain the location of the return address on the current stack frame. ReCFI inserts this intrinsic to locate the return address and check its corresponding taint label before every return instruction. Moreover, ReCFI extends DFSan to properly initialize the taint label of the return address to *untainted* on creation of new stack frames, i.e. in every function prologue.

Although ReCFI instruments function prologue and epilogue, it does not rely on correct stack unwinding. This is because it only inserts taint checking instructions to the epilogue which are read-only. No cleanup instructions are required. Therefore, there are no compatibility issues when the applications use `setjmp/longjmp` or other related functions for exception handling.

Avoiding Taint Labels on the Stack. DFSan implements an optimization strategy to minimize its performance overhead, which introduces a weakness that could be utilized by an attacker to circumvent our control-flow integrity checks: for each local variable, DFSan allocates space for a taint label on the stack. Every time the variable's taint label needs to be checked or updated, DFSan accesses the taint label allocated on the stack instead of accessing the shadow memory region. This optimization saves a few instructions to calculate the shadow address. However, if attackers manage to overflow a stack variable, they might be able to overwrite the value of the taint label and reset it to *untainted*. In this case, our CFI checks could miss the integrity violation and the attack would not be detected. To avoid this potential security breach we removed this optimization.

A second optimization strategy of DFSan concerns propagation of taint labels through function calls. By default, taint labels of arguments and return values are passed through Thread-Local Storage (TLS). To reduce the number of memory accesses, DFSan provides an option to pass taint labels as additional function call arguments alongside their corresponding original arguments in CPU registers. However, for more than 3 parameters (more than 6 in total with taint labels), taint labels would be passed by pushing them to the stack, leading to the same potential security weakness as the first optimization strategy described above. Therefore, we disabled this optimization for ReCFI.

Input Tainting and Collection of Attacker Information. DFSan provides the basic API for input tainting. Application developers can use it to create new taint labels

[5] `https://llvm.org/docs/LangRef.html#llvm-addressofreturnaddress-intrinsic`

and assign them to buffer variables that are considered entry points for attacker-controlled input data. Additionally, DFSan maintains a global map structure `dfsan_label_info` which maps each 16-bit taint label identifier to a pointer of arbitrary application-specific data. With ReCFI, this application-specific data can be used to provide detailed information about the source of the tainted input data, e.g. the source address of a received network packet or the fingerprint of a TLS certificate associated with the current communication session. The more detailed those information, the more precise an attacker can be identified. In case ReCFI detects a CFI violation, it invokes an attack reaction routine and passes taint label information and the associated application-specific data as arguments. We implemented this attack reaction routine in ReCFI's runtime library as a function with `weak` linkage. In this way, application developers can redefine the function symbol to implement their custom attack reaction routine and perform application-specific reactions based on the provided information that ReCFI extracts from the taint labels.

5 Case Study: NginX Web Server

To confirm ReCFI's ability to detect control-flow hijacking attacks on complex programs, we conducted a case study with the popular web server NginX. Versions 1.3.9 and 1.4.0 of NginX contain a stack buffer overflow vulnerability (CVE-2013-2028), which can be exploited to gain remote code execution (RCE). The exploit can be triggered by sending specifically crafted HTTP requests. The vulnerability is caused by an unsound cast from signed to unsigned integer, misinterpreting the size of the network buffer as overly large. Thus, too much bytes are written to the buffer, leading to an overwrite of the return address.

According to ReCFI's Step 0, we first classified the network interface as non-trustworthy and searched for input buffers into which data is read from the network socket. We assigned the initial taint label to network input buffers as shown in Listing 1. We added a call to `dfsan_set_label` to taint the bytes starting at the address of `buf` up to `buf + n`, which equals the number of bytes read from the network socket. The taint label is represented by `c->recfi_label`. It is initialized during connection establishment with the HTTP client and associated to the object `c`, representing the connection. As shown in Listing 2, we modified the function `ngx_event_accept` to insert taint label initialization after NginX accepts the connection request.

With these few manual modifications to NginX, we compiled it with ReCFI which automatically inserts additional code for Steps 1-3. Running the exploit[6], ReCFI successfully detects the corruption of the return address and is able to track the origin of the attack back to the IP address of the client who started the exploit.

For Step 4, we defined a custom attack reaction function directly in the NginX source code, overwriting ReCFI's default attack handler with an application

[6] `https://dl.packetstormsecurity.net/papers/general/nginx_exploit_documentation.pdf`

Listing 1: Instrumentation of NginX's function `ngx_unix_recv()`. Additional instructions for initial input tainting are highlighted.

```
ssize_t
ngx_unix_recv(ngx_connection_t *c, u_char *buf, size_t size)
{
  [...]
  do {
    ssize_t n = recv(c->fd, buf, size, 0);
    dfsan_set_label(c->recfi_label, buf, n);
    [...]
  }
}
```

Listing 2: Instrumentation of NginX's function `ngx_event_accept()`. Additional instructions for initial input tainting are highlighted.

```
void ngx_event_accept(ngx_event_t *ev) {
  [...]
  s = accept(lc->fd, (struct sockaddr *) sa, &socklen);
  struct sockaddr_in *sa_in = (struct sockaddr_in*) sa;
  char *ip = inet_ntoa(sa_in->sin_addr);
  c->recfi_label = dfsan_create_label("accept", ip);
  [...]
}
```

specific reaction as described in Section 4. We implemented a custom attack handler to which ReCFI passes the attacker's IP address. The attackers IP is then blacklisted by adding a new *Linux iptables* rule. This is only an example for attack reactions possible with NginX. If NginX is configured for client authentication via username & password or X.509 certificates, ReCFI could also associate taint labels with the username or the client certificate's fingerprint and blacklist them.

To measure the performance overhead of ReCFI in NginX we used the WRK benchmark[7]. In server software, compute time only makes up a fraction of the overall performance because I/O operations are much more time-consuming. To account for this, we measured the throughput of served HTTP requests within 30 seconds for different file sizes. Results are shown in Table 1. Averaging over all file sizes (geometric mean), ReCFI introduces a slowdown of 1.56x.

Table 1: Slowdown of ReCFI in NginX.

File Size	1 KB	10 KB	100 KB	1 MB
Slowdown	2.10x	1.88x	1.34x	1.11x

6 Security Discussion

In this section, we discuss security guarantees and limitations of ReCFI.

Accuracy in Attack Detection. ReCFI uses dynamic analysis to track the data-flow along actual execution paths. In contrast to CFI approaches solely based on static analysis, there is no over- or under-approximation of the control-flow graph. Hence, there are no false positives when detecting an overwrite of a jump or return address. Therefore, all potential attacks detected by ReCFI are indeed overwrites of code pointers with attacker-supplied input. Assuming every attacker-controllable interface is correctly tainted in Step 0, ReCFI also does not have false negatives, i.e., it is able to detect every overwrite of code pointers with attacker-controlled input data. Accidental overwrites from benign interfaces are not detected because these interfaces are not marked as taint source in Step 0. This is acceptable, because we can assume that they are very unlikely and happen accidentally and thus are a one-time event as opposed a targeted, repeated attack with the intention to exploit the program.

Bounded Number of Taint Labels. As described in Section 4, our implementation of ReCFI can create up to $2^{16} - 1$ distinct taint labels. If applications generate new taint labels very frequently, this can eventually cause a wrap-around of the 16-bit taint label identifier. This could lead to confusion in associating taint

[7] `https://github.com/wg/wrk`

labels with their corresponding taint source (potential attack origin). However, ReCFI still prevents the application from exploitation.

Protection of Taint Metadata. With ReCFI, we adapted DFSan to avoid storing taint labels on the stack as described in Section 4. Therefore, attackers with *contiguous write* capabilities cannot overwrite the security-critical taint metadata. Hence, ReCFI successfully protects programs against attackers complying to our attacker model.

Protection of Programs with External Libraries. Because ReCFI is a compiler-based approach, it can only protect those parts of a program which have been compiled with it. It cannot detect overwrites of code pointers if they happen in external libraries, which *have not been* compiled with ReCFI. In order to effectively protect those parts of the program which *have been* compiled with ReCFI, we have to ensure correct propagation of taint labels even through calls to functions of external libraries. This issue is solved by DFSan which automatically generates a wrapper function for each external library function. Within each wrapper function, the actual external function is called and the taint labels of arguments and return values are propagated according to the data-flow semantic of the external function. DFSan provides a so called *ABI list*[8] which can be used by developers to conveniently specify the data-flow semantic for each external library function.

7 Performance Evaluation

We measured the performance impact of ReCFI using the CPU-bound SPEC CPU2017 benchmark suite[9]. Since we implemented ReCFI primarily for the C programming language, we tested all SPEC CPU2017 benchmarks written in C, omitting those written in C++ and Fortran. All measurements were taken on an Intel Xeon E7-4850 v3 CPU with 2.20 GHz.

Run-Time Overhead. To understand the performance overhead of ReCFI and separately DFSan, we measured both versions and compared their overheads to the baseline compiled with default compiler flags. We ran each benchmark 3 times and used their medians to calculate the overhead relative to the baseline. The resulting run-time overheads are shown in Figure 3. The slowdown caused by ReCFI ranges from 1.99x for the `nab` benchmark to 7.30x for the `perlbench` benchmark, averaging at 3.52x (geometric mean). For comparison with other published approaches and related work, we list their run-time overheads in Section 8.

As depicted in Figure 3, enabling DFSan introduces most of the performance overhead. There are several factors that cause its decrease in run-time performance: (1) An additional memory access to read or write the corresponding taint

[8] `https://clang.llvm.org/docs/DataFlowSanitizer.html`
[9] `https://www.spec.org/cpu2017/`

label for every memory load or store operation, (2) two additional memory accesses to write and read function arguments passed through TLS, (3) taint label union operations for arithmetic instructions, (4) register spills due to additional operations on taint labels, (5) displacement of application memory/code from cache lines in favor of taint label handling. On top of that, ReCFI's additional control-flow integrity checks add (1) an additional memory access to initialize the taint label of the return address to *untainted*, (2) an additional memory access to load the taint label of code pointers before return- and indirect call/jump instructions, and (3) the related instruction to check this label for taintedness. The measurement results show that the overhead introduced by ReCFI on top of DFSan is comparatively small, which endorses our approach for software projects where DFSan is already used.

For the benchmarks `perlbench` and `gcc`, ReCFI and DFSan introduce larger overheads, presumably because both benchmarks test program compilation workloads which execute many memory load and store operations while the other benchmarks are more bound to integer and floating point arithmetic. For some benchmarks, e.g., `lbm` or `nab`, the measurement results suggest that ReCFI has smaller overhead than DFSan. We assume that for those benchmarks, ReCFI does not have a significant impact on performance compared to DFSan, and that ReCFI's additional memory access instructions have a positive effect on internal caching structures of the CPU, leading to minor improvements in the overall run-time compared to DFSan.

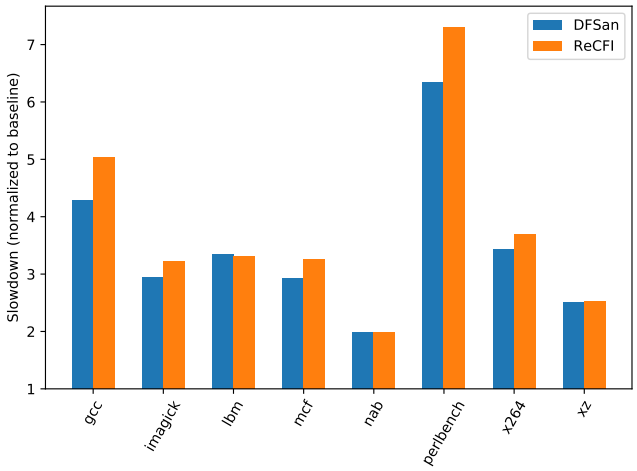

Fig. 3: Normalized run-time overhead for SPEC CPU2017.

Memory Overhead. During our run-time benchmarks, we also measured the peak memory usage of every benchmark program by monitoring their maximum

resident set size. The average memory overhead measured for ReCFI is 3.07x. As ReCFI does not allocate any memory on top of DFSan, there is no significant difference from DFSan's memory overhead. It is caused by the following factors: (1) two additional bytes shadow memory for every byte of application memory, (2) the DFSan union table, and (3) the `dfsan_label_info` map as described in Section 4. The `dfsan_label_info` structure is a map that associates 20 byte of additional data and pointers per taint label identifier and thus uses $2^{16} \times 20B \approx 1.3MB$ of memory. DFSan's union table UT is a two-dimensional array implementing a strictly triangular matrix which tracks whether two taint labels i and j have been combined to a new union taint label k as result of an arithmetic instruction with two tainted operands.

$$UT[i][j] \mapsto k, \text{ with } i < j \quad \text{(strictly triangular)}$$

With taint labels being 16 bit in size, a full union table would consume approximately 4.2 GB of memory.

$$\sum_{k=1}^{2^{16}-1} (k) \times 16bit \approx 2.1 \times 10^9 \times 2\,\text{B} \approx 4.2\,\text{GB}$$

However, in practice only few distinct taint label combinations exist in the union table. Therefore, only few of the virtual memory pages containing the union labels are actually mapped to physical RAM. Thus, the physical memory usage for the `dfsan_label_info` map and the union table is constant while shadow memory additionally consumes 2x application memory. Hence, programs like the SPEC CPU2017 benchmarks, which have application memory consumption of multiple gigabytes, have approximately tripled memory usage with ReCFI.

8 Related Work

Within the last 10 years there were many publications in the field of control-flow integrity. Most of the classical compiler-based CFI approaches only introduce overheads of 1.01x-1.45x [3] but are subject to false negatives [4] and completely lack the ability to gain information about the attack origin. Therefore, we will omit them from our summary of related work and refer to [3] as a comprehensive overview of recent development in that field. In the following, we focus on proposed exploit mitigation techniques that utilize dynamic information-flow tracking to gain insights about the attacker which can be used to prevent repeated attack attempts. We classify them based on their instrumentation techniques.

Hardware-Based. Early approaches were based on architectural modifications (e.g. to CPU or MMU) to track data-flow and check for tainted return/jump addresses in hardware [15,5,10,8]. Dedicated hardware clearly brings the advantage of low run-time overhead. Reported slowdowns range between <1.01x

to 1.17x for compute-bound benchmarks. However, implementations are based on non-commodity hardware prototypes that are not widely available, greatly inhibiting their adoption. They do not need access to applications' source code but often require modifications to the operating system, further decreasing practicability. ReCFI is a compiler-based approach which can run on commodity CPUs and only requires minimal changes to the application code to mark potentially attacker-controlled interfaces as taint source.

Dynamic Binary Instrumentation. Chronologically, hardware-based approaches have been superseded by approaches based on Dynamic Binary Instrumentation (DBI). With DBI, an application's binary code is loaded and executed by a binary interpreter. During execution, it can analyze and modify program code on-the-fly to add instructions for taint propagation and monitoring of code pointers. This technique is very flexible and does not require recompilation. However, analysis and instrumentation during run-time introduce more overhead than ReCFI in most cases. The authors of TaintCheck [14] (based on the *Valgrind* framework[10] reported a slowdown of up to 40x. *Dytan* [7] and *libdft* [11] both utilize the Intel PIN framework [12]. Their respective authors reported 50x slowdown and 7-11x slowdown. TaintTrace [6] (based on DynamoRIO[11]) proposes multiple performance optimizations, the most impactful of which is using a direct shadow mapping between application memory and taint metadata similar to our approach. The authors measured a slowdown of 5.5x on SPEC2000 INT benchmarks.

Static Instrumentation. Static instrumentation techniques have been proposed to eliminate overheads of DBI for inserting additional instructions at run-time. Instead, they insert all additional instructions before the program is executed. Xu et al. [17] accomplish this by utilizing the CIL framework [13] for *Source-to-Source Transformation* of C code to insert additional instructions. They report a slowdown of 1.76x on average for compute-bound benchmarks. However, for performance optimization they use stack variables to store the taint metadata of the protected application's stack variables. This introduces new security vulnerabilities in cases of stack buffer overflow attacks. ReCFI actively avoids that as we described in Section 4. Moreover, building on source-to-source transformation, Xu et al. only protect applications written in C while ReCFI is a compiler-based approach and therefore is able to protect programs of all languages that compile to LLVM IR.

The most recent work related to ReCFI is *Iodine* [2]. It is also a compiler-based static instrumentation approach and is built on top of DFSan. Iodine reduces run-time overhead by applying *Optimistic Hybrid Analysis (OHA) [9]* to taint tracking. They use a profile-guided static analysis to eliminate some of the additional taint checking instructions on the fast-path and fall back to conservative dynamic taint checking on the slow-path. Banerjee et al. report an average

[10] `https://valgrind.org/`
[11] `https://dynamorio.org/`

slowdown of 1.41x for SPECint C benchmarks. While they focus mainly on performance optimizations for generic information-flow tracking problems, ReCFI specializes on detection of code pointer overwrites and generating information about the attack origin. Hence, they are complementary approaches.

9 Conclusion

In this paper, we presented ReCFI, a compiler-based security mechanism to detect control-flow hijacking attacks and gain information about the attack origin by utilizing dynamic taint analysis. Our system statically instruments applications at compile-time to insert additional instructions for taint propagation and integrity-checking of return addresses, function pointers and indirect jump targets. Our implementation is based on LLVM's Dataflow Sanitizer (DFSan), an actively maintained framework for dynamic data-flow analysis. ReCFI can be used by enabling compiler-flags of the clang C compiler and only requires minimal modifications to the source code by developers to identify potentially attacker-controlled interfaces as taint source. We extended DFSan with security measures to protect against attackers with contiguous write capabilities. Utilizing dynamic analysis, ReCFI precisely tracks the data-flow of the program without over- or under-approximation and therefore does not yield false positives nor false negatives as opposed to classical compiler-based CFI approaches [4]. We outlined how ReCFI can be used in server software to track an attack back to the attackers' IP addresses and exclude them from further communication. ReCFI introduces a slowdown of 3.52x on average for SPEC CPU2017 benchmarks and only 1.56x on average for the real-world web server NginX. Thus, ReCFI advances the state of the art for control-flow integrity solutions with attack origin identification.

Acknowledgments This work is partially funded by the Bavarian Ministry of Economic Affairs, Regional Development and Energy.

References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. pp. 340–353. CCS '05, Association for Computing Machinery, New York, NY, USA (Nov 2005)
2. Banerjee, S., Devecsery, D., Chen, P.M., Narayanasamy, S.: Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 490–504. IEEE, San Francisco, CA, USA (May 2019)
3. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-Flow Integrity: Precision, Security, and Performance. ACM Computing Surveys **50**(1), 16:1–16:33 (Apr 2017)

4. Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In: 24th USENIX Security Symposium. pp. 161–176 (2015)
5. Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., Iyer, R.K.: Defeating memory corruption attacks via pointer taintedness detection. In: 2005 International Conference on Dependable Systems and Networks (DSN'05). pp. 378–387 (Jun 2005)
6. Cheng, W., Qin Zhao, Bei Yu, Hiroshige, S.: TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In: 11th IEEE Symposium on Computers and Communications (ISCC'06). pp. 749–754 (Jun 2006)
7. Clause, J., Li, W., Orso, A.: Dytan: A generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. pp. 196–206. ISSTA '07, Association for Computing Machinery, New York, NY, USA (Jul 2007)
8. Dalton, M., Kannan, H., Kozyrakis, C.: Raksha: A flexible information flow architecture for software security. In: Proceedings of the 34th Annual International Symposium on Computer Architecture. pp. 482–493. ISCA '07, Association for Computing Machinery, New York, NY, USA (Jun 2007)
9. Devecsery, D., Chen, P.M., Flinn, J., Narayanasamy, S.: Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis. ACM SIGPLAN Notices **53**(2), 348–362 (Mar 2018)
10. Katsunuma, S., Kurita, H., Shioya, R., Shimizu, K., Irie, H., Goshima, M., Sakai, S.: Base Address Recognition with Data Flow Tracking for Injection Attack Detection. In: 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06). pp. 165–172 (Dec 2006)
11. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: Libdft: Practical dynamic data flow tracking for commodity systems. ACM SIGPLAN Notices **47**(7), 121–132 (Mar 2012)
12. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Notices **40**(6), 190–200 (Jun 2005)
13. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Horspool, R.N. (ed.) Compiler Construction. pp. 213–228. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2002)
14. Newsome, J., Song, D.: Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In: Proceedings of the 12th Network and Distributed Systems Security Symposium (2005)
15. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 85–96. ASPLOS XI, Association for Computing Machinery, Boston, MA, USA (Oct 2004)
16. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In: 23rd USENIX Security Symposium. pp. 941–955 (2014)
17. Xu, W., Bhatkar, S., Sekar, R.: Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In: 15th USENIX Security Symposium (2006)