Wrangling in the Power of Code Pointers with ProxyCFI

Misiker Tadesse Aga, Colton Holoday, Todd Austin

Wrangling in the Power of Code Pointers with ProxyCFI

Misiker Tadesse Aga, Colton Holoday, and Todd Austin

University of Michigan, Ann Arbor, USA
{misiker,choloday,austin}@umich.edu

Abstract. Despite being a more than 40-year-old dark art, control flow attacks remain a significant and attractive means of penetrating applications. Control Flow Integrity (CFI) prevents control flow attacks by forcing the execution path of a program to follow the control flow graph (CFG). This is performed by inserting checks before indirect jumps to ensure that the target is within a statically determined valid target set. However, recent advanced control flow attacks have been shown to undermine prior CFI techniques by swapping targets of an indirect jump with another one from the valid set.
In this article, we present a novel approach to protect against advanced control flow attacks called ProxyCFI. Instead of building protections to stop code pointer abuse, we replace code pointers wholesale in the program with a less powerful construct – pointer proxies. Pointer proxies are random identifiers associated with legitimate control flow edges. All indirect control transfers in the program are replaced with multi-way branches that validate control transfers with pointer proxies. As pointer proxies are uniquely associated with both the source and the target of control-flow edges, swapping pointer proxies results in a violation even if they have the same target, stopping advanced control flow attacks that undermine prior CFI techniques. In all, ProxyCFI stops a broad range of recently reported advanced control flow attacks on real-world applications with only a 4% average slowdown.

Keywords: CFG mimicry attacks · CFI · Pointer proxy

1 Introduction

For more than four decades, control flow attacks, in which attackers force programs into executing code sequences not anticipated by the developer, have played an important role in the infiltration of vulnerable systems. These attacks are particularly attractive to attackers because they immediately give them the agency necessary to deploy attack payloads, leak important information, embed a rootkit, launch an additional attack such as privilege escalation, etc. As such, there has been much attention paid to reducing a system's vulnerability to control flow attacks.

Early measures to stop control flow attacks include StackGuard [16], data execution prevention (DEP) [2] [1] and address space layout randomization (ASLR) [3]. However, subsequent attacks have skirted this defenses [35] [27] [34]. CFI [10] follows a principled approach to mitigating control flow attacks by enforcing the runtime execution path of a program to adhere to the statically determined CFG. It does these by checking if the target of an indirect jump is within

a valid set of targets. However, prior proposed CFI solutions are either impractical or ineffective. Some, which strictly follow a program's CFG [28], have high overheads that render them impractical to production systems. Others attempt to reduce overheads by approximating the CFG with limited classes of targets (*e.g.*, two classes for function pointers and return addresses) [41] [42] [6] [4], but these do not protect against control flow attacks that swap targets while remaining on the CFG [22] [14] [20] [33].

In this work, we make the key observation that many of the vulnerabilities in control flow stem from the excessive power inherent in code pointers. To stop the tide of control flow attacks, we propose a novel approach to control flow integrity, called ProxyCFI, that replaces all code pointers in the program with *pointer proxies*. A pointer proxy is a unique random identifier (64-bits in our implementation), which represents a forward or backward control flow edge in the program. Consequently, all indirect jumps in the program (*e.g.*, returns and jumps-through-register) are replaced with multi-way branches that implement a direct jump to the address associated with the pointer proxy. As pointer proxies are a function of both the source and the target of an edge, swapping pointer proxies results in a violation even if they have the same target.

To ensure that all execution flows stay on the program CFG for even third-party ProxyCFI compliant code, a binary-level program verifier first validates at load-time that programs and libraries have CFGs that are fully discoverable, use only pointer proxies, and avoid all indirect jumps/returns. Finally, to thwart attacks based on binary analysis, the verifier re-randomizes pointer proxies at load time. In addition, the loader marks code sections unreadable, to protect from active-read attacks that gather pointer proxies using memory leaks.

Table 1: **Comparison of Code Pointers to Pointer Proxies.** Pointer proxies preserve program control integrity by reducing their capabilities. This table lists the differences in capabilities between code pointers and pointer proxies. Ultimately, it is the powerful nature of code pointers that enable many control flow attacks.

	Code Pointers	**Pointer Proxies**
Arithmetic Allowed	Yes	No
Totally Ordered	Yes	No
Trivial Forgery Attacks	Yes	No
Permit Relative Distance Attacks	Yes	No
Replay Attacks on Returns and fptrs	Yes	Only from the same source address

More importantly, ProxyCFI has a number of powerful features to deter attacks that mimic legitimate control flow (*i.e.*, control flow attacks that seemingly remain on legitimate control flow edges), such as control flow bending (CFB) [14]. These attacks exploits the fact that existing CFI techniques allow executions to maliciously divert indirect branches if the target address is still in the valid set of targets. ProxyCFI thwarts this as a pointer proxy is unique to a particular source and target address which makes a pointer proxy used in one function context invalid in another even if they share the same target addresses.

Table 1 lists the comparative capabilities of traditional code pointers versus pointer proxies. As shown in the table, pointer proxies do not support arithmetic

manipulation; thus, relative-address based control flow attacks, such as ASLR de-randomization attacks [34] would not be possible with pointer proxies. Moreover, pointer proxies are much more difficult to forge, since their assignment is not in anyway related to other pointer proxies, whereas pointer values often reveal much information through relative address distances to other code objects, facilitating relative address inspired attacks. Since pointer proxies are unique to a given function, return address copy attacks, such as the return-into-libc [37] and backward-edge active-set attacks [36], become more challenging, as the pointer proxies of other functions (which are assigned at load-time) must be leaked and then translated to the local function's proxies (which have no correlation even if the current function calls the intended target).

1.1 Contributions of This Paper

In this paper, we introduce ProxyCFI, a novel control flow integrity technology that thwarts recent advanced control flow attacks while incurring low performance overhead. Specifically, this paper makes the following contributions:

- We present ProxyCFI that provides an efficient and practical protection against advanced code reuse attacks, with a threat model notably more capable than that of traditional CFI techniques which must protect a shadow stack [32] or pointer encryption technologies which must protect encryption keys [17].
- We detail the implementation of ProxyCFI within the GNU GCC compiler toolchain.
- We demonstrate the efficiency of the approach running a wide range of CPU-centric and network-facing applications. In addition, we implement two compile-time optimizations, which ultimately reduce the slowdown of this technology to only 4% on average. In addition, our security analysis shows that the technology stops real-world advanced control flow attacks and demonstrates 100% coverage for the RIPE x86-64 control flow attack suite [40].

2 Protecting Control Flow with ProxyCFI

In this section, we detail our threat model and the broad ProxyCFI concept, then present how to build and verify programs (including shared libraries) with pointer proxies.

2.1 Threat Model

In this work, we assume a very powerful attacker who wants to redirect control flow to a code sequence that deviates from the programmer-specified CFG. In accomplishing their control flow attack, the attacker has read and write access to any data location, including globals, stack and heap variables, as well as data storage locations holding pointer proxies. The code segment of the program is assumed to be non-writable.

Given this powerful attacker, ProxyCFI works to prevent the attacker from hijacking control from the programmer-specified CFG. In addition, ProxyCFI also gives protection against non-gadget code reuse attacks (*e.g.*, COOP where the attack does not leave the CFG of the program, but instead enlists the code in a CFG mimicry attack [33]).

2.2 Pointer Proxies

To stop control flow attacks, we replace all code pointers with pointer proxies. A pointer proxy is a random identifier (64-bits in our evaluated implementation), where pointer proxy P represents an edge from a particular code exit point to a code entry point. Wherever a code pointer would reside in the program (*e.g.* in a jump table or on the stack as a return address), it is replaced by its corresponding pointer proxy value P. Figure 1 illustrates a small code snippet in which the code pointers have been replaced by pointer proxies. As seen in the example, where code pointers would have been stored (*e.g.*, on the stack for a return address), they are replaced with pointer proxies (denoted by a $).

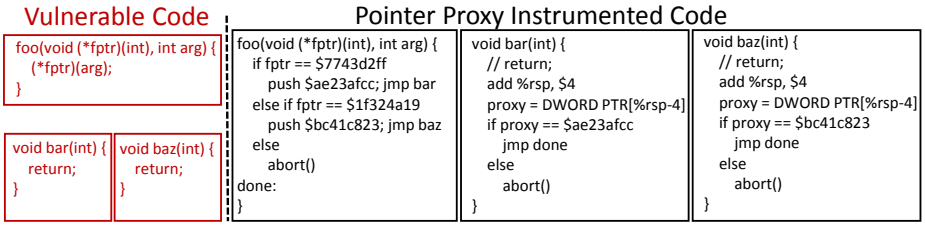

Fig. 1: **Example Code Sequence using Pointer Proxies.** Pointer proxies replace code pointers in a program with random identifiers associated with legal control flow edges. Multi-way direct branches translate pointer proxies into direct jumps.

At indirect jumps and returns, the pointer proxy is inspected, and then using a multi-way direct branch, the appropriate code entry point associated with the pointer proxy is jumped to. We call a multi-way direct branch, which matches a pointer proxy and then directly jumps to the associated code target, a ***sled***. Direct jumps are not replaced with pointer proxies. Since our threat model assumes that code cannot be written, any direct jump is inherently write-protected, and thus no additional protections are required. Three multi-way branches can be seen in the example in Figure 1. The indirect call to *bar()* and *baz()* in function *foo()* is implemented with a multi-way branch that jumps to *bar()* if the proxy *$7743d2ff* is encountered and jumps to *baz* when the pointer proxy is *$1f324a19*. Additionally, both of the returns from functions *bar()* and *baz()* are implemented with a multi-way branch.

Advanced control flow attacks such as control low bending [14] undermine CFI by using a code pointer copied from one function context to jump to addresses in some other function without violating CFI constraints. Pointer proxies are uniquely assigned to control flow edges (*i.e.*, a function of source and destination), thus, the pointer proxies of function X are meaningless to function Y. This powerful feature, which does not impact the usability of pointer proxies, thwarts large number of advanced control flow attacks. This aspect is shown in Figure 1 in the returns of functions *bar()* and *baz()*. While both functions return to the same address (*i.e.*, label *done*), they each use a distinctly different pointer proxy. As such, if each of the functions were to disclose each other's pointer proxy and return upon it, it would not match any target in the return's multi-way branch, and would result in a violation. To stop the potential forgery

of pointer proxies, all pointer proxy values are defined per-function, and they are re-randomized at program load time by the verifier as detailed in Section 2.4.

2.3 Building Code with Pointer Proxies

Building code to work with pointer proxies requires replacing every place in the program that uses a code pointer with a pointer proxy.

Hardening indirection with pointer proxies: All indirect branches (*e.g.*, switch jump tables, indirect calls, and returns), are replaced with a multi-way direct branch sleds. Of course, to know what targets must be tested for in a sled, we must fully anticipate all of the targets of each indirect branch. For locally sourced indirect jumps, such as a switch statement jump table, we can easily anticipate in the current compilation module all of the indirect jump targets. Indirect calls and returns require more analysis as the locations (and pointer proxy values) may come from another application module (*e.g.*, a shared library). Consequently, the compilation framework must support whole-program CFG analysis (including the call graph). In our prototype implementation in the GNU GCC toolchain, we utilize a two-pass compilation strategy, to first build the whole program CFG and then to compile programs with fully enumerated multi-way direct branch sleds. Details of this compilation strategy are covered in Section 3.

At indirect calls, the type of the target function is noted, and when the whole program CFG is constructed, an indirect function call is assumed to possibly happen to only same-typed function and has had its address taken. Similarly, the return address sleds target the instruction after all actual and potential calls to the returning function, where a potential call directly targets the function or indirectly targets the function through a compatible function pointer. This approach works quite well until a program declares a *void ** indirect function call, which could potentially call any function in the program. Fortunately, our optimizations detailed in Section 3.2 perform well to reduce the overall impact of these generic indirect function calls. To ensure that there is a valid sled entry for code pointers used across modules, ProxyCFI performs whole program analysis including shared libraries to ensure indirect calls from a shared library into the executable have a corresponding sled entry to handle callback functions. A similar analysis is performed for callbacks passed into shared libraries.

When good code pointers go bad: ProxyCFI doesn't support dangerous code pointer operations such as code pointer arithmetic, which are characteristics of a buggy program [29]. For example, we could manufacture a code pointer to a private function in x86-64 GCC by simply adding the size of the preceding function (in bytes) to its code pointer. In our prototype GNU GCC C/C++ implementation of ProxyCFI, we issue a security warning for these operations at compile time and terminate the program if executed at runtime. While we were able to create these problems in test programs, none of our benchmark programs suffered from dangerous code pointer manipulations.

setjump() and *longjmp()* require special handling in the compiler because these functions implement a unique user-directed program control flow transition. Together, these functions implement a superset of function pointer behavior, such that a call to *setjmp()* can be the target of any other *longjmp()* in the

program. Both function pointers and *longjmp()* share an indirect jump, but a *longjmp* should generate a multi-way direct branch including all of the pointer proxies assigned to the instruction immediately after each call to *setjmp()*. Thus, any tampering in the *setjmp()* control context cannot pull execution off the CFG.

2.4 Load-time Program Verifier

Load-time Program Verifier ensures no legal control flow results in a violation and no deviation from the CFG is missed. It maintains this by verifying that programs utilize only pointer proxies for indirect branches via multi-way direct branches that are fully enumerated by the programmer-specified CFG. If an unexpected pointer proxy is encountered, the program is terminated.

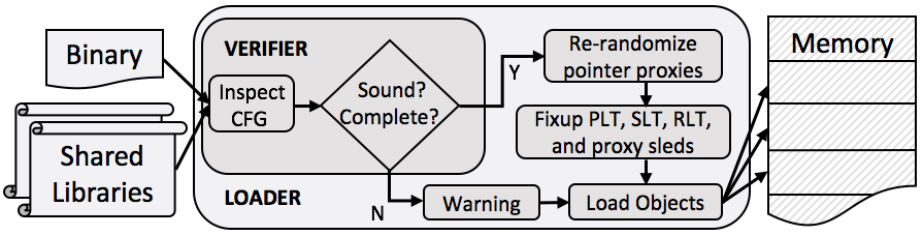

Fig. 2: **ProxyCFI Program Loader.** This figure illustrates the process of loading a ProxyCFI compliant program for execution.

Figure 2 shows how a binary or shared library is loaded and validated. The verifier ensures that only pointer proxies are used for control transitions. If a code object passes verification, the verifier generates load-time assigned pointer proxies, such that an attacker cannot anticipate any pointer proxy values even with an active read attack on the program.

The psuedocode for the ProxyCFI code verifier is shown in Algorithm 1. The verifier performs reachability analysis on the code object's CFG to validate that it is *i)* free of indirect control transfers and *ii)* all control transfers point to a valid instruction within the current code object or the entry point of another code object for calls. To this end, it performs a depth-first traversal of the CFG of the code object, inspecting all control transfer instructions. If indirect call/jump or return instructions are encountered in the code object, it immediately fails verification. For direct control transfer (*i.e.*, direct call, direct jump, loop instructions), it analyzes the target address for any possible violations.

For direct jump instructions, the verifier checks that the target address points to a valid instruction within the current code object. For direct function calls, the verifier validates that the target is a valid code object entry point. Finally, for multi-way branch sleds replacing an indirect call/jump and return, the verifier ensures that all targets are valid according to the CFG. Once the verifier completes reachability analysis of the CFG without failure, the code is safe to load and execute.

Algorithm 1 Load-time Program Verifier. The algorithm performs a reachability analysis of the CFG to identify any illegal jumps or uses of indirection.

```
 1: procedure VERIFY(obj)
 2:     for all f in obj do
 3:         ep ←f.entry_point
 4:         while ep ≠ ∅ do
 5:             e ← ep.pop()
 6:             if e.checked == True then
 7:                 continue
 8:             else
 9:                 br ←scan_for_next_branch(e)
10:                 switch br do
11:                     case Indirect(br) or Invalid(br.target)
12:                         return Fail
13:                     case Direct_Branch
14:                         ep.push({br.target, e.next})
15:                     case sled
16:                         inspect ({sled.proxies})
17:                         ep.push({sled.targets})
18:     return Success
```

2.5 Deterring CFG Mimicry Attacks

A mimicry attack [38] on the CFG is one that implements attacker-directed control *without* leaving the programmer-specified CFG. With the introduction of powerful control flow integrity mechanisms, such as CFI [10] these non-gadget code reuse-based attacks have quickly grown in number, including counterfeit OOP [33], control flow bending [14], and active-set backward-edge attacks [36]. ProxyCFI can provide protection against these attacks through per-function pointer proxy namespaces and load-time pointer proxy assignment.

Per-function pointer proxy namespaces: Traditional full-fledged code pointers represent a code location that is sharable with any other part of the program. It is this property that allows an adversary to copy a code pointer from one function and replay it in another, an approach that Counterfeit OOP [33] utilizes to implement method-level code reuse that does not leave the CFG. Pointer proxies deter CFG mimicry attacks by defining unique pointer proxy namespaces for each function. Thus, if a function copies the pointer proxy from another function, for example, by searching for pointer proxies up the stack, any attempt to use it will always result in a violation when the multi-way branch sled executes. In Figure 1 although *bar*() and *baz*() both return to the same location, they each have their own proxy namespace which have different proxy-to-edge mappings (i.e., $ae23afcc and $bc41c823).

Load-time assignment of pointer proxies: Pointer proxies are re-randomized at load time to further deter mimicry attacks. This prevents offline analysis of code to generate a translation table from pointer proxies to source and target code addresses. Enforcing a non-readable code section and load-time assignment of proxies significantly complicates CFG mimicry attacks.

2.6 Shared Libraries with Pointer Proxies

Shared libraries are an attractive target for control flow attacks as they are used among multiple applications. Attacks that target libc, for example, can be reused

on any application that links to it. The classic attack is *return-into-libc* [37], wherein the adversary overwrites the return address so that the program returns to an exploitable *libc* function such as *system()*. In addition, most shared libraries contain large enough codebases that leaves an attacker with wide selection of gadgets for all classes of code reuse attacks, such as ROP [18], JOP [12], LOP [25] and their variants [13] [21].

Connections into and out of shared libraries must be managed by unshared data or code that is generated dynamically. Returns and indirect calls are natural solutions to entering and exiting shared libraries because they draw on unshared data in the stack and the global offset table, respectively. As such, shared libraries require special handling with ProxyCFI which works to remove indirection. One solution to securing shared libraries is to forbid them – Intel chose this with SGX [23]. However, we want to retain their advantages: modularity, reduced page swapping, and simplified version management.

ProxyCFI compliant shared libraries use *unshared code* to manage control flow in and out because indirection must be replaced with multi-way branches. While calling a shared library function still goes through the *procedure linkage table (PLT)*, the indirect call within the PLT is dynamically replaced with a pointer proxy sled. At load time, extra space in the caller's address space is mmap'ed for the code that channels control flow on return. Shared library functions have their returns statically replaced with a relative jump down to the unshared multi-way branches.

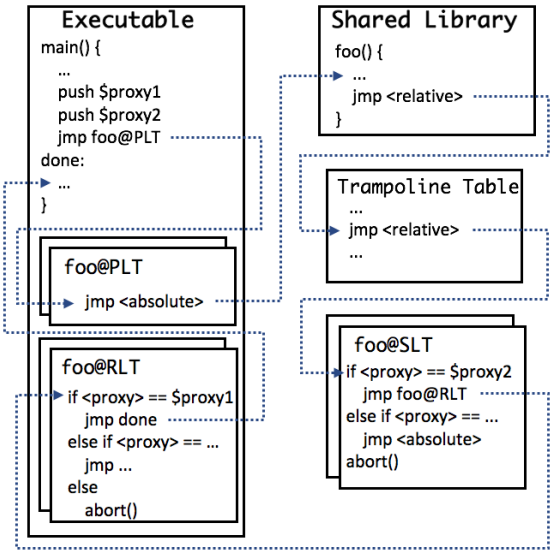

Fig. 3: **Shared Library Control Flow.** ProxyCFI allocates linkage tables using pointer proxies in the caller's address space, which permit safe entry and exit from a compliant shared library.

Our approach to deploying shared libraries with ProxyCFI is illustrated in Figure 3. We split the process of returning from a shared library into two stages, which are associated with the *selection linkage table (SLT)* and the *return linkage table (RLT)*, respectively. Two pointer proxies are used to return from a shared

library, one for the SLT and one for the RLT. For each PLT entry to a shared library function there is an RLT entry that contains a multi-way branch leading back to each call site in the code object. If shared library function $foo()$ is called from multiple code objects, then each code object would have a separate RLT entry for $foo()$ in its own address space. While the RLT specifies how to return within a code object, the SLT specifies which code object to return to. To accomplish this, SLT entries contain a multi-way branch of absolute jumps directed at RLT entries. Since the size of SLT entries varies based on the code objects that use the shared library, a trampoline table facilitates static generation of the relative jumps by forwarding control onto the appropriate SLT entry.

Using load-time assignment, it is possible to assign proxies for the tendrils into a shared library in a way that creates pointer proxies when the library is first loaded. Moreover, our approach allows the pointer proxies used to enter and exit the shared library function to be unique to each address space that utilizes the shared library. This ensures that an attacker cannot gather pointer proxy information from their own address space and use it to attack a program using the same shared library.

3 ProxyCFI in GNU GCC

In this section, we detail the implementation of ProxyCFI in the GNU GCC C/C++ toolchain. We present the overall compilation flow, and then dive into the details of the optimizations implemented.

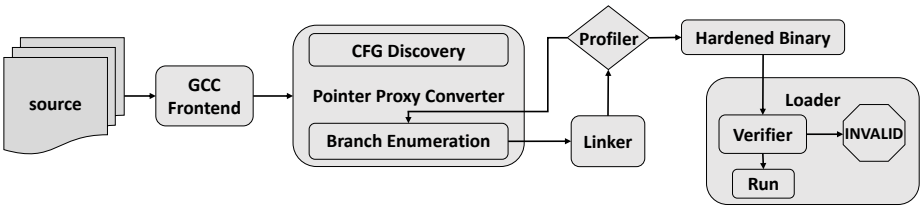

Fig. 4: **Compilation Flow.** The compilation occurs in two passes. In the first pass, the entire CFG of the program is discovered. In the second pass, all legal program entrypoints are assigned randomly selected pointer proxy identifiers, and all indirect jumps, indirect calls, and returns are replaced with fully enumerated multi-way direct branches. The linker resolves all jumps using compiler-generated global identifiers for all entry points. The profiler instruments the code to count the most frequent targets of multi-way branches, which is used for optimization. Finally, all code is passed through the pointer proxy verifier, assigned load-time random pointer proxies and loaded into execute-only pages before execution begins.

3.1 Compilation Flow

ProxyCFI instrumentation is done with a two-pass transformation on assembly generated mid-compilation by the existing GCC infrastructure. All sites of indirection are replaced with fully enumerated multi-way direct branches that validate CFG transitions with pointer proxies. Figure 4 describes the overall flow of ProxyCFI compilation.

– **Pass 1. CFG Discovery**: Assembly files are parsed for function labels, (direct or indirect) call sites, and return sites. Return edges are constructed by observing the target set for each direct and indirect call. Indirect call target sets include only functions that have had their addresses taken and have a matching type signature. Type information on function pointer calls are passed from the GCC frontend to the ProxyCFI compiler core.
– **Pass 2. Branch Enumeration**: Since the CFG contains all transitions between functions, multi-way branch targets are fully enumerated before the second pass begins. For return sites, pointer proxies are chosen in the context of the called function and shared with the calling function's indirect call sled. Pointer proxies are generated in this way to deter CFG mimicry attacks (see Section 2.5 for details).

After generating a binary, runtime analytics, which are generated by the profiler, are passed back to the branch enumeration phase, at which point the multi-way branch sleds are rewritten with optimizations. Load time invocation of the verifier rewrites all pointer proxies before executing the program.

3.2 ProxyCFI Optimizations

Indirect jump and return sleds can become very long, especially for frequently called functions. To address these potential concerns, we implemented two optimizations: profile-guided sled sorting, and function cloning.

Profile-guided sled sorting. The main source of performance degradation with ProxyCFI is the overhead incurred by the repeated comparisons used to implement multi-way direct branch sleds. The number of checks required is directly proportional to the number of legitimate targets for the corresponding indirect control transfer instruction. Yet, we observed that these sleds were highly biased to only a few of the branch targets. Our sled sorting optimization takes advantage of the biased distribution of multi-way branch targets by sorting the entries in descending order of profiled execution count. As shown in Section 4, this optimization significantly reduces the average depth a program must traverse into a sled before finding the pointer proxy target.

Function Cloning. While profile-guided sorting of the sleds significantly reduces performance degradation associated with multi-way branch sleds, the improvements are limited for functions with more uniformly distributed sled profiles. To combat this, we adapted function cloning [15] – an optimization that creates specialized copies of functions – as a means to reduce overall sled lengths. For sleds with more uniform distributions, this optimization significantly reduces the performance overhead incurred by executing sleds. Figure 5 illustrates function cloning. A function with near uniform sled distribution is cloned (*e.g.*, function *f2* becomes identical functions *f2* and *f2_c1*). Then, half of the call sites to the cloned function are redirected to the cloned function.

This optimization also significantly reduces the attacker's agency in selecting CFG edges to exploit for CFG mimicry attacks [14] [20] [33].

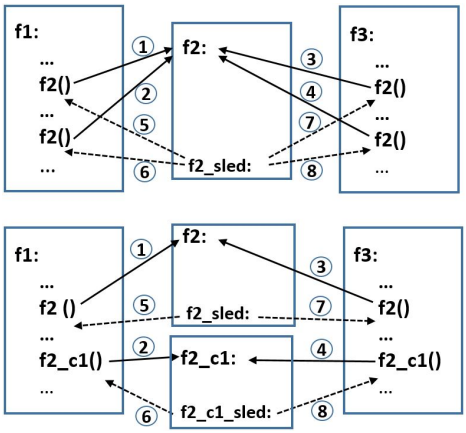

Fig. 5: **Function Cloning.** Function cloning cuts the number of legitimate edges by a factor of the number of clones. Legitimate edges 6 and 8 from f2_sled are no longer legitimate after cloning.

4 Evaluation

In this section, we examine the performance and security of ProxyCFI. First, the performance impact of ProxyCFI is assessed by examining the slowdown incurred for many CPU-centric and network-facing benchmarks, with and without ProxyCFI optimizations. To gauge the security benefits, we performed penetration testing with the RIPE control flow attack suite [40] and recent advanced control flow attacks on real-world applications.

4.1 Evaluation Framework

ProxyCFI build framework. Our ProxyCFI compiler framework was built on GCC version 6.1.0. In our evaluations, we used Ubuntu 16.04 on x86-64. Using x86-64 is essential in our implementation because our shared libraries rely heavily on relative jumps to preserve code page sharing, which is significantly more efficient in 64-bit x86. We customized *GLIBC's* loader to handle ProxyCFI compliant shared libraries and mark code pages execute-only. Many modern processors have hardware support for execute-only memory. For example, recent Intel CPUs support unreadable code pages using the Memory Protection Keys (MPK) feature.[1] In our prototype implementation, we used this feature to make the code section execute-only (*i.e.*, disabled read/write access).

Benchmarks analyzed. We evaluated the performance and space overhead incurred by ProxyCFI using the SPEC CPU 2006 benchmarks. In addition, we evaluated the overhead on the network-facing application ***redis-server***, running it with the standard *redis-benchmark* with 50 parallel clients and a 3-byte payload. To isolate the performance overhead incurred by ProxyCFI-hardened shared objects, we also ran microbenchmarks for varying shared library sled depths. To evaluate the security guarantees provided by ProxyCFI, we analyzed

[1] Execute-only memory is also supported on ARMv8 and above.

applications from all the major categories commonly targeted by control flow hijacking attacks including multimedia processing, Javascript engines, document rendering, network infrastructure and VM interpreters. Specifically, we analyzed the following commonly attacked applications (detailed in section 4.3):
MuPDF is a light weight PDF XPS and EPUB parsing and rendering engine. MuPDF versions V1.3 and prior have a stack-based buffer overflow vulnerability (CVE-2014-2013) [8] that results in remote code execution via a maliciously crafted XPS document.
bladeenc is a cross-platform MP3 encoder which is also used as a daemon for encoding in distributed MP3 encoders/CDDB servers like *abcde*. *bladeenc* has several vulnerabilities that lead to CFG mimicry attacks that could be exploited remotely (CVE-2017-14648) [7].
dnsmasq is a DNS forwarder designed to provide DNS services to a small-scale networks, and it is included in most Linux distributions. Versions of *dnsmasq* prior to 2.78 have a stack-overflow vulnerability which enables a remote attacker to send a maliciously crafted DHCPv6 request to hijack control flow on the target system (CVE-2017-14493) [5].
Gravity is a dynamically typed concurrent scripting language written in C. The Gravity runtime contains a stack-based buffer overflow that leads to remote code execution (CVE-2017-1000437) [9].

4.2 Performance Analysis

We ran the SPEC CPU 2006 benchmarks performance analysis experiments on an Intel Xeon Gold 6126 Processor with 24 cores and 32GB RAM running Ubuntu 16.04 LTS Xenial Xerus

Figure 6 shows the performance overhead incurred by ProxyCFI instrumentation. For compute-intensive applications, the näive implementation's performance overheads are non-trivial, since these programs have high average sled depth. Average sled depth is a measure of how many pointer proxy tests are required in a sled, on average, before a direct branch is taken. Ideally, we would like this value to be close to 1 to lower the performance overhead for ProxyCFI. For applications with heavy use of function calls such as *perlbench*, *gobmk* and *sjeng*, the performance degradation for the unoptimized implementation is more pronounced, having a average return sled depth of 27 for *perlbench*.

With optimizations, the average sled depth drops dramatically, as do the performance overheads. For example, *perlbench* benefits significantly from profile-guided sled sorting optimization. *h264ref* also benefits significantly from optimizations, as it makes heavy use of generic function pointers with indirect functional call sleds having up to 855 entries, of which only two are frequently targeted. *gcc*, on the other hand, makes considerable use of both function calls (average sled depth of 32) and generic function pointer (with an average sled depth of 26 for indirect calls). The performance benefit of the function cloning optimization is more visible on *gcc*, as the probability distribution of taken branches falls off slower than the other applications. For network-facing applications, the performance overhead is insignificant due to their I/O-bound nature. The average performance overhead for *redis-server* is 0.25% and an average overhead of 0.93% for all of network-facing applications we evaluated.

Figure 7 shows the percentage increase in the binary size as a result of pointer proxy instrumentation, both with and without optimizations. On average the

code size grows by 49% for our benchmarks with the worst case of 121% for *h264ref*, due to the large amount of instrumentation required for its generic function pointers. Finally, Figure 8 shows the impact of ProxyCFI verification and load-time proxy randomization on program load times. As shown in the graph, it has approximately linear relationship with code size, the longest being 1200ms, consistent with previous works that perform load-time randomization [39] [30].

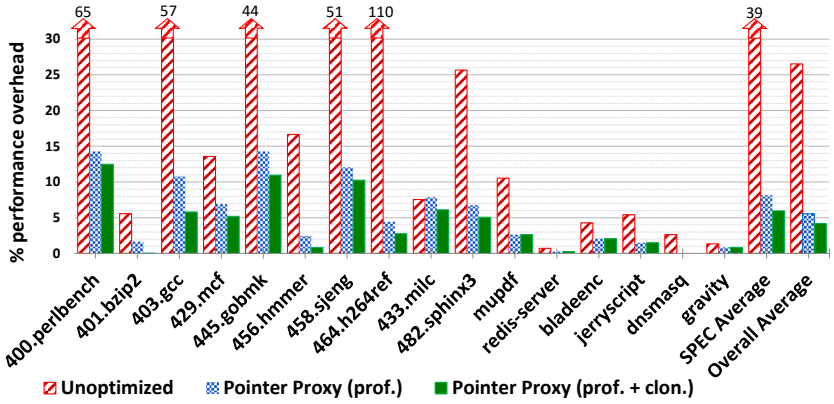

Fig. 6: **Performance Overhead of ProxyCFI.** Unoptimized shows the performance overhead without any optimization, while ProxyCFI (prof.) and ProxyCFI (prof. + clon.) show performance with profile-guided sled sorting (prof.) and function cloning (clon.) optimizations.

Shared library performance. We measured the cost of our shared library support infrastructure by microbenchmarking entries and exits to shared libraries and comparing it against unprotected shared library calls. The average percent slowdown for a shared library calls using optimized ProxyCFI compilation is 1.48% and 2.31% respectively for the best and worst-case average sled hit depths observed in our benchmark experiments.

4.3 Security Analysis

To assess the security strength of ProxyCFI, we first examine its ability to stop control flow attacks in the RIPE attack suite, then we examine to what extent ProxyCFI can stop real-world control flow attacks including CFG mimicry attacks.

Penetration testing with RIPE: RIPE is a control flow attack testbed that generates attacks by permuting five dimensions of attack: location (*e.g.*, stack, heap, ...), target (*e.g.*, return address, function pointers, ...), overflow technique (*e.g.*, direct/indirect), and function of abuse (*e.g.*, memcpy, ...) [40]. Native RIPE targets 32-bit x86 code, thus, with the help of a recently implemented low-fat pointer extension [19], we ported the RIPE test suite to x86-64. Our port supports the following five dimensions: location, target (including *setjmp()* and *longjmp()*), method, and overflow type. Permuting all RIPE dimensions totals

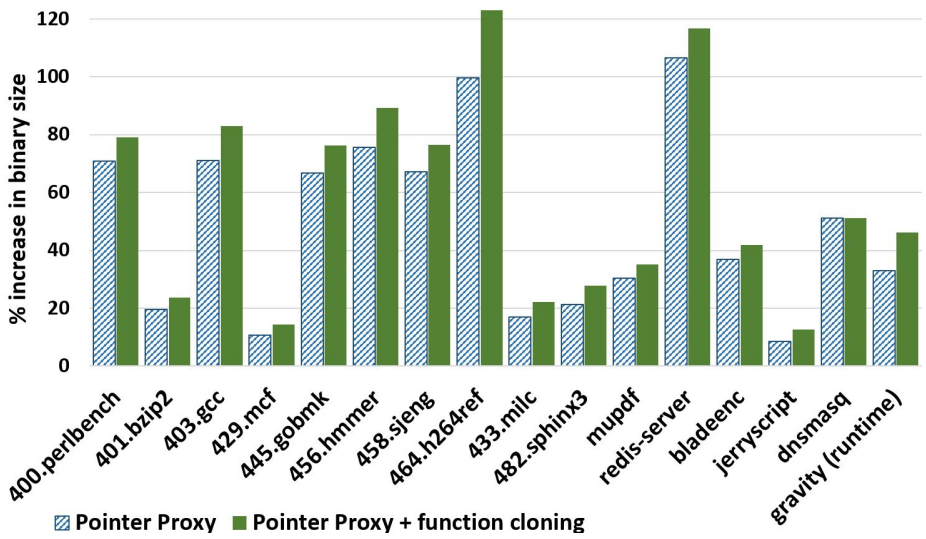

Fig. 7: **Increase in Code Size.** This graph shows the impact of ProxyCFI on code size. The blue bars (left) represent unoptimized ProxyCFI programs, while the green bar (right) represents optimized ProxyCFI programs.

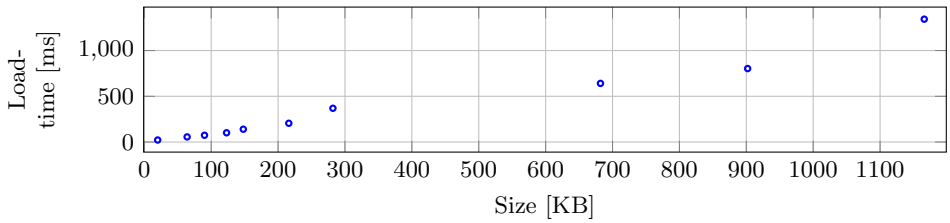

Fig. 8: **Load-time overhead vs. code size** This graph shows the impact of pointer proxy randomization on load-time.

up to 850 unique tests. **With ProxyCFI protections, 100% of the RIPE attacks are stopped.** In addition, ProxyCFI was able to detect the exact point at which attacks escape the CFG.

Real-world vulnerabilities. To evaluate the effectiveness of ProxyCFI against attacks on real-world applications, we included recent attacks reported on the National Vulnerability Database (NVD) in our evaluation.

With ProxyCFI, we were able to stop all of the following real-world attacks including CFG mimicry attacks.

In testing, we found that the declared violations enabled us to quickly identify the root cause of the vulnerability. We analyzed four attacks.

MuPDF has a stack-based buffer overflow vulnerability in the *xps_parse_color*() function which performs an unchecked *strcpy()* of a user supplied (via XPS input) array to a fixed size buffer [8]. The exploit uses this bug to overwrite the return address and jump to an ROP gadget. With ProxyCFI we were able to detect the stack pivot based on the corrupted pointer proxy.

Bladeenc's command line parser uses unchecked calls to *strcpy()* to copy parameters to a 256-byte buffer that are exploited for arbitrary code execution by using a carefully crafted command line arguments [7]. The exploit corrupts a function pointer to jump to another function which is also in its legal target set to hijack control flow via a CFG mimicry attack. We were able to detect the exploit when trying to jump using a forged pointer proxy (which was interpreted as invalid pointer proxy from the source address).

Dnsmasq has a vulnerability caused by an unchecked use of *memcpy()* in the *dhcp6_maybe_relay()* function to a 16-byte field of the variable *state*. This bug allows an attacker to perform inter-object overflow to perform ROP attack. Using ProxyCFI we were able to detect all of the exploits.

Gravity contains a stack-based buffer overflow in the function *operator_string _add*() which can be used to write past the end of a fixed-sized static buffer to achieve code execution. The exploit uses this vulnerability to overwrite a return address using a malicious Gravity script. For the ProxyCFI hardened version the attack was detected when the exploit tried to make an indirect jump based on forged pointer proxy.

5 Related Work

Memory safety. Memory corruption attacks have been often used to hijack control flow, either by injecting code or reusing existing code. Data execution prevention [1] [2] is sidestepped entirely as reuse attacks need not inject code. Comprehensive memory safety techniques such as Softbound [31] can completely eradicate memory exploitation, but they suffer from high overhead or compatibility issues, deeming them as yet impractical for widespread adoption.

Control flow integrity: A new wave of practical defenses emerged with a focus on validating that execution adheres to a static, programmer specified CFG. Control flow integrity (CFI) [10] was the first of these CFG defenses. The defense inserts checks before indirect branches to make sure that all indirect control transfers are within the statically discovered CFG. Various coarse-gained variants have relaxed CFI constraints to achieve practical solutions through both software and hardware approaches [41] [42] [6] [4].

CCFIR [41] uses a load-time randomized springboard section to redirect all indirect control flow transfers, which has been bypassed by a successive work [22]. Intel CET [6] provides rudimentary hardware protection for forward edges through its indirect branch tracking. Microsoft CFG enforces a weak form of CFI by restricting indirect function calls to function entry points [4]. While these techniques are valuable against straightforward code reuse techniques, CFG mimicry attacks effectively bypass this CFI techniques. Unlike these coarse grained CFI techniques, ProxyCFI provides fine grained protection, and also affords protection against CFG mimicry attacks. CCFI [28] is a fine-grained CFI technology that protects code pointers by storing hash based message authentication code (MAC) alongside code pointers and checking the MAC before indirect branches. While CCFI can protect against CFG mimicry attacks, its high performance overhead (52% for SPEC'06) will undoubtedly limit its applicability in production environments. Like CCFI, ProxyCFI provides fine-grained control flow protection, while incurring significantly lower overheads (only 5.9% average slowdown for SPEC'06).

Other control flow integrity works have proposed to completely remove instructions employed for control flow hijacking attacks. Return-less kernels [26] avoid use of *ret* instruction by replacing them with a lookup into a static return table which provides protection solely against return-based attacks. Control-data isolation (CDI) [11] rewrites both forward and backward edges with exclusively direct branches. CDI would conceivably constrain execution to the programmer-specified CFG, if it were to verify that all binaries adhered to CDI compilation requirements. But since the approach still uses code pointers to identify program pointers, the approach is readily attackable with control flow attacks that do not leave the CFG, such as Counterfeit OOP [33]. Moreover, ProxyCFI addresses CFG mimicry attacks by replacing code pointers with pointer proxies that utilize per-function namespaces, which are assigned at program load-time to an execute-only memory.

Code-Pointer Integrity (CPI) [24] provides memory safety for code pointers by storing them in a safe region. CPI requires allocation of a safe data region inaccessible to an attacker. ProxyCFI does not require any special data region protections.

6 Conclusion

While significant effort has been spent to shut down control flow attacks, their existence and value persists today, even 40 years after the first buffer overflow attack. With ProxyCFI, we take the novel approach of replacing all of a program's code pointers with the much less powerful pointer proxy. A pointer proxy is a random identifier representing a specific program entry point from the context of a specific function. A control transfer with a pointer proxy utilizes a multi-way direct branch which fully anticipates all of the potential jump targets. As such, ProxyCFI provides much resistance to advanced control flow attacks because it is difficult to forge/swap pointer proxies to mimic a legitimate CFG transition. Our implementation of ProxyCFI is built into the GNU GCC C/C++ compiler toolchain, such that all code pointers are replaced with pointer proxies including those contained within shared libraries. Analysis of our pointer proxy implementation reveals that they introduce minimal slowdown when pointer-proxy specific optimizations are applied, only an average 4% slowdown across a wide range of benchmarks. Moreover, security analysis of ProxyCFI shows that it stops all of the control flow attacks we tested, including 100% of the attacks in the RIPE x86-64 attack suite and a wide range of real-world attacks including CFG mimicry attacks.

Looking ahead we see a number of avenues for growing the capabilities of ProxyCFI. In particular, we would like to implement support for reassigning pointer proxy values at runtime, and we would like to explore the use of pointer proxies for a limited set of data pointers.

Acknowledgement

This work was supported by DARPA under Contract HR0011-18-C-0019. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

1. Data execution prevention, 2003. Accessed: 2018-02-29.
2. Linux kernel 2.6.8, 2004. Accessed: 2018-02-29.
3. Windows isv software security defenses, 2010. Accessed: 2018-02-29.
4. Control flow guard (windows) - msdn - microsoft. https://msdn.microsoft.com/en-us/library/dn919635.aspx, 2015. Accessed: 2018-04-13.
5. Cve-2017-14493. https://www.cvedetails.com/cve/ CVE -2017-14493/, 2017. Accessed: 2018-02-12.
6. Intel control-flow enforcement technology (cet). https://software.intel.com/sites/default/files/managed /4d/2a/control-flow-enforcement-technology-preview.pdf, 2017. Accessed: 2018-04-13.
7. Bladeenc: Vulnerability statistics. https://www.cvedetails.com/product/2851/Bladeenc-Bladeenc.html, 2018. Accessed: 2018-01-05.
8. Cve-2014-2013. https://www.cvedetails.com/cve/ CVE-2014-2013/, 2018. Accessed: 2018-04-13.
9. Cve-2017-1000437. https://www.cvedetails.com/cve/ CVE-2017-1000437/, 2018. Accessed: 2018-01-05.
10. Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.
11. William Arthur, Ben Mehne, Reetuparna Das, and Todd Austin. Getting in control of your control flow with control-data isolation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 79–90. IEEE Computer Society, 2015.
12. Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.
13. Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.
14. Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., 2015. USENIX Association.
15. Keith D Cooper, Mary W Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, 1993.
16. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.
17. Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.
18. Dino Dai Zovi. Practical return-oriented programming. *SOURCE Boston*, 2010.
19. Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. 2017.
20. Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios 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*, pages 901–913. ACM, 2015.

21. Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 417–432. USENIX Association, 2014.
22. E. Gktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, May 2014.
23. Intel. Dynamic libraries, 2015. Accessed 2018-02-29.
24. Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, 2014. USENIX Association.
25. Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. Loop-oriented programming: a new code reuse attack to bypass modern defenses. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 190–197. IEEE, 2015.
26. Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208. ACM, 2010.
27. Limin Liu, Jin Han, Debin Gao, Jiwu Jing, and Daren Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 37–44. IEEE, 2011.
28. Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 941–951. ACM, 2015.
29. C+ MISRA. Guidelines for the use of the c/c++ language in critical systems. *MIRA Limited. Warwickshire, UK*, 2012.
30. Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque control-flow integrity. In *NDSS*, volume 26, pages 27–30, 2015.
31. Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices*, 44(6):245–258, 2009.
32. Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.
33. Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.
34. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.
35. Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8. ACM, 2009.
36. Michael Theodorides and David Wagner. Breaking active-set backward-edge CFI. In *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*, pages 85–89. IEEE, 2017.
37. Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

38. David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.
39. Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.
40. John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM, 2011.
41. Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.
42. Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *USENIX Security Symposium*, pages 337–352, 2013.

A Redis-benchmark Results Breakdown

Table 2 shows the results of running *redis-server* with the standard *redis-benchmark* using 50 parallel clients and a 3-byte payload.

Table 2: Results of running *redis-benchmark* ProxyCFI compliant *redis-server* versus unhardened baseline

Command	**Baseline** (request/sec)	**ProxyCFI** (request/sec)
PING_INLINE	12 320.9	12 115.34
PING_BULK	12 881.67	12 926.58
SET	12 469.83	12 158.05
GET	12 941.73	13 010.67
INCR	10 514.14	11 189.44
LPUSH	12 227.93	12 997.47
RPUSH	11 592.86	11 828.72
LPOP	12 659.83	12 255.46
RPOP	12 804.1	12 604.8
SADD	12 218.96	12 055.46
HSET	12 023.57	11 872.7
SPOP	11 855.36	11 552.39
LPUSH (needed to benchmark LRANGE)	12 968.49	12 600.12
LRANGE_100 (first 100 elements)	6506.82	6325.19
LRANGE_300 (first 300 elements)	2788.99	2690.05
LRANGE_500 (first 450 elements)	2403.4	2211.26
LRANGE_600 (first 600 elements)	1730.2	1652.59
MSET (10 keys)	10 409.08	9959.79