ρFEM: Efficient Backward-edge Protection Using Reversed Forward-edge Mappings

Paul Muntean
Technical University of Munich
paul.muntean@sec.in.tum.de

Matthias Neumayer
Technical University of Munich
matthias.neumayer@tum.de

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Gang Tan
The Pennsylvania State University
gtan@psu.edu

Jens Grossklags
Technical University of Munich
jens.grossklags@in.tum.de

Claudia Eckert
Technical University of Munich
claudia.eckert@sec.in.tum.de

ABSTRACT

In this paper, we propose reversed forward-edge mapper (ρFEM), a Clang/LLVM compiler-based tool, to protect the backward edges of a program's control flow graph (CFG) against runtime control-flow hijacking (*e.g.,* code reuse attacks). It protects backward-edge transfers in C/C++ originating from virtual and non-virtual functions by first statically constructing a precise virtual table hierarchy, with which to form a precise forward-edge mapping between callees and non-virtual calltargets based on precise function signatures, and then checks each instrumented callee return against the previously computed set at runtime. We have evaluated ρFEM using the Chrome browser, NodeJS, Nginx, Memcached, and the SPEC CPU2017 benchmark. Our results show that ρFEM enforces less than 2.77 return targets per callee in geomean, even for applications heavily relying on backward edges. ρFEM's runtime overhead is less than 1% in geomean for the SPEC CPU2017 benchmark and 3.44% in geomean for the Chrome browser.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; • **Software and application security**;

KEYWORDS

Clang/LLVM, control flow integrity, hijacking attack, cyber defense.

ACM Reference Format:
Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2020. ρFEM: Efficient Backward-edge Protection Using Reversed Forward-edge Mappings. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3427228.3427246

1 INTRODUCTION

Control-flow hijacking attacks, such as return oriented programming (ROP) [1], have threatened software systems for a long time. These attacks proliferate mainly due to the fact that statically building a precise program control flow graph (CFG) is practically not feasible. Building the CFG requires precise program alias analysis of, *e.g.,* source code, binary, intermediate representation (IR), which is not tractable due to its undecidability [2]. Consequently, the obtained CFG is an over- or under-approximation of the real program's CFG. In addition to control flow forward-edge (*i.e.,* `jump`, `call`) violations, backward-edge (*i.e.,* `ret`) violations play a crucial role in facilitating attacks.

In general, there are two main approaches to protect the integrity of backward edges during runtime: (1) check-based approaches including μRAI [3], PittyPat [4], CFL [5], PT-CFI [6], τCFI [7], which check if the function `ret` instruction targets the legitimate return address; and (2) stack-discipline-based approaches including SafeStack [8], RAD [9], Microsoft's RFG [10], Zieris *et al.* [11], Shadesmar [12], BinCFI [13], GCC's ShadowStack [14], and double stacks [15]. However, most shadow stack techniques rely on information hiding for security. Unfortunately, information hiding (disclosure) based defenses are generally vulnerable [3] to information disclosure [16, 17], profiling attacks [18] and to at least four other attacks [19], which can be used independently to bypass shadow stacks.

In this paper, we seek to design an alternative to shadow stack techniques. We present ρFEM, a compile-time software instrumentation tool used to enforce a fine-grained CFI-based policy for protecting backward edges against control-flow hijacking attacks. The key idea is to use a precise compiler-generated program virtual table hierarchy to protect virtual callee targets, and a precise reverse forward-edge function signature mapping to protect non-virtual callee targets. While previous work has adopted such an approach to protect backward edges either through function signatures [20, 21] or through class hierarchy analysis [22–24], we are not aware of any other purely source code based solution that combines the two approaches for a comprehensive protection. In terms of instrumentation, in contrast to [25], which uses instruction prefetch, we use: (1) `NOP`s allowing us to encode `ID`s into them; (2) multiple checks for covering cases where the same function can be invoked directly and indirectly; and (3) ID intervals which make the checks more efficient.

A key advantage of ρFEM over previous tools (*e.g.,* information hiding) is that its instrumentation is write-protected at all times, does not rely on information hiding, which is fundamentally broken from a security perspective [12], does not use special purpose registers/segments, which may not be present on all types of systems, and its metadata primitives can be easily re-purposed to protect forward edges.

We have implemented ρFEM atop the Clang/LLVM [26] compiler framework and evaluated it with a set of real-world programs including all pure C/C++ programs contained in the SPEC CPU2017 benchmark. ρFEM has a low runtime overhead with them, while maintaining high calltarget return address precision. Further, as *e.g.,* AIR [13], fAIR [21], and AIA [27] are not ideal (according to Carlini

ACSAC 2020, December 7–11, 2020, Austin, USA

et al. [28]) as these capture only average target-reduction results, we use the *RTR* [29] metric to provide more detailed results and insights.

In summary, we make the following contributions:

- We design a novel fine-grained backward-edge protection technique that relies on reversed forward-edge mappings.
- We implement our technique based upon the Clang/LLVM compiler framework inside a prototype called ρFEM[1].
- We evaluate ρFEM thoroughly and report a runtime overhead of 3.44% in geomean for the Chrome browser, less than 1% in geomean for the SPEC CPU2017 benchmark, and 2.77 return targets per callee in geomean for the tested programs.

2 BACKGROUND

2.1 Virtual Table Hierarchy

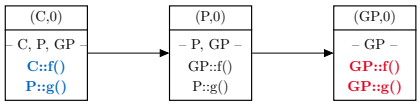

Figure 1: Virtual tables (vtable) of a single C++ class hierarchy.

Figure 1 depicts a single derived vtable type *C* with a parent vtable *P* and grandparent vtable *GP* (*i.e.*, the root class of both functions *f()* and *g()*). An arrow depicts a parent-child class inheritance relation. The arrow tip indicates the parent vtable whereas the other arrow end indicates the child vtable. With regard to a specific virtual callsite (which is used to call a virtual function contained in this vtable hierarchy), we introduce the following definitions.

Precise class. The precise class of a callsite is the least-derived type of which the object used at the callsite can be. Usually, the precise class is the static type of the variable used for the virtual call. In our example, we assume that we have a virtual callsite which uses a variable of static type *C*. This static type is the precise class of the callsite.

Base class. We define the base class as the class which provides the function implementation which is called when an object of precise type is used, *i.e.,* the object has a dynamic type of precise class. Therefore, the vtable entry used for the object dispatch is located in the vtable of precise class and, per definition, points to a function of base class. It follows that if the precise class itself implements (or overrides) the function used at the callsite, then the precise class and the base class of the callsite are the same. Figure 1 depicts in blue shaded color the vtable entries used, in case the object at the callsite is of precise type (*C*). If the callsite dispatches function *f()*, then the base class for this callsite is *C*, since *C* overrides function *f()*. If instead the callsite dispatches function *g()*, then the base class for this callsite is *P*, because class *C* does not override function *g()* and instead uses the implementation provided by class *P*.

Root class. The root class is defined as the class first introducing the function (*i.e.,* the least-derived class declaring the function). Note that this class might declare this function as `abstract` and not provide any implementation for it.

2.2 Backward Control Flow Edges

The control flow of a program can be captured using a control flow graph (CFG), in which the forward edges represent the function call or (un)conditional jumps, and backward edges represent the return. The return transfers in assembly code are usually represented by return instructions and are used to return the control flow of the program to the address after the callsite which originally called the function. Depending on the instruction set architecture (*e.g.,* x86, x86-64, ARM, SPARC), the format of the return instruction can vary (*e.g.,* `ret` in x86/x86-64). Note that for the herein mentioned return-edge details, the format of the corresponding instruction is irrelevant. However, the pre-conditions and post-conditions needed for normal program execution are of importance. These are determined by the used calling convention and can slightly vary depending on the used architecture binary interfaces (ABIs) (*e.g.,* Itanium [30], Microsoft [31], ARM [32]).

2.3 Shadow Stack Techniques

General Description. Shadow stack techniques are used similarly to stack canaries, in order to protect against backward-edge program control-flow hijacking attacks. These techniques consist of complementing the program with additional code, which is able to check if the caller/callee function calling convention is respected during runtime. The technique relies on building a second stack for each function stack located in the program. Runtime checks ensure that each function return address, which was put in the shadow stack before entering the context of the called function, is popped from the stack before leaving the called function or before the stack frame was cleaned up by the program. Essentially, a shadow stack technique keeps track of all addresses that are pushed and popped on the stack and checks that the push-pop address pairs match. This way, the caller/callee function calling convention is enforced. In this fashion, the program stack is checked to be not corrupted (polluted) by the attacker with fake addresses, that are usually used to chain code reuse gadgets, as for example in return oriented programming (ROP) attacks. While these techniques are effective in theory, they have received only partial acceptance. For example, SafeStack [8] is in production, but was recently bypassed; see Goktas *et al.* [19] for more details. Lastly, their effectiveness is influenced by the information hiding technique on which most rely. As previously mentioned, most shadow stack techniques rely on information hiding for security. An exception is "Read-only RAD" [9], which uses `mprotect` to keep the shadow stack pages read-only except when needed for updates. Likewise, hardware-based shadow stacks (*e.g.,* Intel CET [33]) also do not rely on information hiding. Next, we list some significant properties of shadow stack techniques.

Hiding the Shadow Stack. The software based shadow stack is typically hidden from the usual program execution through one level (ideally more) of program indirection (*i.e.,* trampolines, segment register, *etc.*). The goal of these levels of indirection are to guarantee that the attacker is not able to find the shadow stack which remains at all time writable. This ensures that the attacker cannot locate and write into the stack (it resides in writable memory) its own return addresses. Further, it is not yet demonstrated, that one level of indirection is sufficient to ensure that the shadow stack cannot be found (through information leaks) by a motivated and resourceful attacker. Shadow-stack implementations put the shadow stack in writable memory; accordingly, if found, it could be overwritten by attacker-controlled addresses in order to mount an attack. As a consequence, approaches which split the shadow stack in two parts have been presented. These approaches [11] essentially separate the sensitive data in yet another

[1] ρFEM Source code: https://github.com/TeamVault/rhoFEM

memory area. As such, if the main shadow stack is compromised the sensitive data remains hidden from the attacker.

Program Binary Size. Shadow stack-based techniques provide a separate shadow stack frame for each function, that is either instrumented inside the protected program or inside a library loaded along with the protected program. Research on shadow stack techniques (see Dang *et al.* [15]) reports a negligible increase in the size of the program binary (on disk or in memory). Even for parallel shadow stacks, the memory usage is modest; and for traditional shadow stacks, it is indeed negligible, since the return addresses are stored compactly. However, these techniques might not be suitable for all types of restrictive program memory applications, such as certain embedded devices where the program heap is small in size. Lastly, note that shadow-stack techniques, which are, for example, based on hardware features such as Intel's CET [33] (Intel *Tiger Lake* based CPUs, which contain CET, are currently on the market) and compiler support, have negligible program binary size blow-up. The binary blow-up of hardware based (*e.g.,* CET) and software based shadow stacks will be similar; CET will still need pages of memory to store the shadow stack. The advantage of CET is that the shadow stack will be fast, and content write-protected.

Special Callee Types. The `C` and `C++` programming languages, for which most of the shadow stack techniques were designed, provide some function calls, that do not return or respect the caller-callee function calling convention, such as: `longjump`, `tail calls`, *etc.* For these types of calls, the shadow-stack techniques reach their technical limitations, since these types of function calls do not return to the address next to the callsite. Lastly, currently available compilers enforcing shadow stack policies do not handle these types of calls: (1) due to complexity reasons, and (2) because these calls do not violate the caller/callee function calling convention, as these do not return at all.

Runtime Overhead. As each function return address has to be pushed, compared with the stack top value, and popped from the shadow stack, the runtime overhead varies drastically from one shadow stack technique implementation to another depending on how efficient this process is implemented. Depending on the count of operations (instructions), which need to be performed (1-3), some research-based shadow techniques have high performance overheads (around 10%; see Dang *et al.* [15] for more details), making them infeasible for deployment in production software. For these reasons, researchers have looked for approaches to do these operations with a minimal number of steps (see GCC's and Clang's shadow stack implementations for more details), such that the overhead is as low as possible and no memory leaks are generated.

Support for External Calls. Most `C/C++` based programs rely on third-party libraries; as such, calls to functions residing in these external libraries can be made. For this reason, this type of external call needs to be protected as well. However, note that most of the research-based binary approaches and compiler-based shadow stack approaches do not protect these shared libraries for the following reasons. First, binary-based tools usually cannot deal with functions having their address not taken. Second, binary-based tools often fail to analyze large binaries due to their complexity. Third, the compiler-based tools opt to not recompile shared libraries due to increased analysis complexity, thus backward edges (also forward edges) remain unprotected. For example, BinCFI [13] (binary tool) could have easily added a shadow stack to protect libraries, but omitted it, due to the resulting overhead. In other words, the backward edges contained in shared libraries are not protected and accordingly the attack surface remains high, and the protection added to the program does not help considerably when all the needed gadgets reside in a shared library.

Emulated Shadow Stacks. Techniques approximating a perfect shadow stack do not contain all caller/callee address pairs. Further, these approaches (*e.g.,* [34]) are mostly based on CPU features such as from/to address pairs and achieve only a coarse-grained precision w.r.t. the return addresses, that need to be checked. This is because these techniques are optimized for performance, and some of the return edges remain unprotected, due to their imprecision. Furthermore, the checks of harvested addresses are slow due to: (1) the high volume of data flowing through the CPU, (2) the need to collect and analyze this data, and (3) the relatively low speed of the continuous reads. As such, these techniques are mostly inefficient against attacks which use backward edges (see Schuster *et al.* [35]). Therefore, other techniques, which are more fine-grained or have a comparable precision as a perfect shadow-stack implementation should be used instead.

3 THREAT MODEL AND ASSUMPTIONS

Adversarial Capabilities.

- *System Configuration.* In our threat model (*e.g.,* STRIDE [36]), we assume that the adversary is aware of the applied defenses and has access to the source and non-randomized binary of the target application.
- *Vulnerability.* We assume that the target program suffers from a memory corruption vulnerability (*e.g.,* `C++` object type confusion [37]) that allows the adversary to corrupt memory objects in order to read from and write to any arbitrary memory address.
- *Scripting Environment.* Further, the attacker can exploit a scripting environment to process memory disclosure information at runtime, adjust the attack payload, and subsequently launch a code reuse attack.

Defensive Requirements.

- *Writable (xor) Executable Memory.* The target system ensures that memory can be either writable or executable, but not both at the same time, *e.g.,* DEP [38]. This prevents an attacker from injecting new code or modifying existing executable code.
- *JIT Protection.* Next, we assume that mitigations are in place to prevent code injection into the just-in-time (JIT) code cache and prevent reuse of JIT-compiled code [23]. These protections are orthogonal to ρFEM.
- *Brute-forcing Mitigation.* Lastly, we require that the protected software does not automatically restart after hitting a booby trap that terminates execution. For example, in the Web browser context, this may be accomplished by displaying a warning message to the user and closing the offending process.

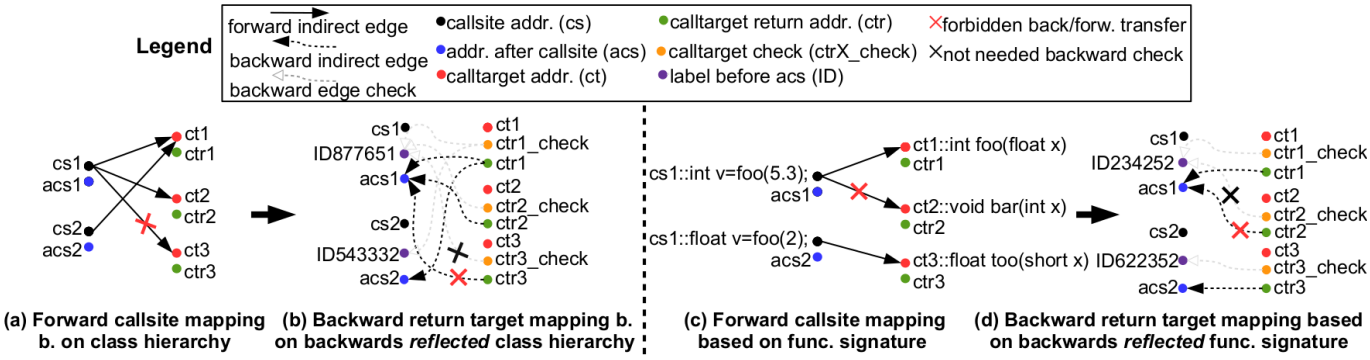

Figure 2: Identifier (ID) based backward-edge mapping of virtual functions called through object dispatches, (a) & (b), and ID based backward-edge mapping of non-virtual functions called indirectly through function pointers, (c) & (d). Essentially, for each virtual callee (virtual function) the program class sub-hierarchy relationship is enforced between callee and corresponding callsite. This is achieved by assigning the same ID at the callee check (before returning) and at the matching callsites. Lastly, for each non-virtual callee (non-virtual function), a function signature is computed which allows only matching callers to call matching callees. This is done by inserting the same ID inside the callee check (before returning) and at the matching callsites.

4 OVERVIEW

4.1 Objectives and Scope

Objectives. During control flow hijacking attacks, attackers often corrupt the backward edges, *e.g.,* by overwriting return addresses. The goal of ρFEM is to protect the backward edges of control flow transfers. While state-of-the-art approaches are using shadow stacks, they suffer from information leakage attacks. Therefore, we seek to design an alternative to shadow stack techniques in order to protect backward edges. Inspired by the initial CFI design in which IDs are used to check the integrity of both forward and backward edges, we extend its ID checks with more precise information, especially by considering the sophisticated caller-callee relation among virtual calls and indirect calls.

Scope. In general, it is a hard problem to determine the point-to relation among pointers especially at the binary code level. We assume we have the source code of the to-be-protected program, and thus we want to perform source code analysis in order to instrument and demonstrate the feasibility of our approach. Currently, ρFEM does not aim to protect tail calls, position independent code, and long jumps. Further, we assume that these program primitives do not allow the attacker to access usable gadgets which may help to craft an attack.

4.2 Approach

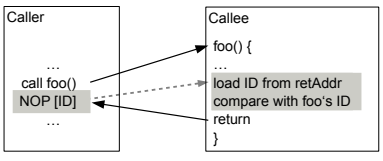

Figure 3: Loading callsite's ID.

Figure 3 shows the instrumentation inserted by ρFEM at both caller and callee sites. In particular, the NOP instruction is inserted right after the caller (left box), while the runtime check is inserted before the callee returns (right box). The black shaded arrows represent the forward-edge and backward-edge transfers, while the gray shaded arrow represents the load of the NOP payload before the callee return.

The presented technique consists of assigning a unique ID (identifier) to every function. Then, every callsite is annotated with the IDs of the functions it legitimately can call. Before returning, the callee retrieves the ID from the callsite and compares it with its own ID (or ID range). This helps to ensure that the callee is only allowed to return to the addresses located after a callsite, which can call this callee. In order to store and later retrieve an ID from a callsite, ρFEM inserts these in x86 NOP instructions located right after the call instruction. Further, ρFEM uses the argument of this NOP instruction to store the ID. The checks contained inside the callee as well as the callsite NOPs are inserted at compile-time and reside in the code section of the program due to their placement inside the functions. Therefore, these cannot be overridden or disabled at runtime as this code is loaded into non-writeable memory.

After a virtual callsite, two NOP instructions are used, as due to inheritance a range of possible callees is allowed (see Virtual Function Analysis for more details). For function pointer based callsites, a function signature approach is used (see Indirect Call Analysis for more details). Since the caller-callee convention is enforced by the compiler, the return address of the callee should point to the next instruction after the callsite. At this instruction new NOPs will be inserted.

The actual check is done inside the callee's function body, right before the RET instruction is executed. This check will load and extract the ID from the NOP using the return address found on the stack. In case the extracted data matches the ID of the callee, the check passes and the RET instruction is executed and the execution continues. In this case, the same return address located on the stack is used to jump to and the program execution continues from that particular address. Otherwise, if the extracted IDs do not match, we assume that the return address stored on the stack was tampered with and we, therefore, assume that a backward-edge violation happened. Thus, program execution is stopped in order to prevent a potential attack.

4.3 Mapping Backward Edges to Callsites

ρFEM fills the NOPs located at the caller and callee with IDs which help to construct a mapping between legal callees and callers. This mapping is used to enforce CFI policies between legitimate callees and callers. To achieve this, ρFEM builds two separate caller-callee mappings: (1) for non-virtual functions (callees), and (2) for virtual functions (callees). Next, we present how these two mappings (sets) for each caller are constructed.

Figure 2 depicts how backward-edge return address sets are built based on the program class hierarchy in sub-figures (a) and (b) for virtual callsites, and based on a precise forward-edge function signature type mapping in sub-figures (c) and (d) for pointer-based function callsites.

Function Signature Based (for non-virtual callees). Mapping backward edges using function signatures (see Figure 2(c) forward-edge mapping and Figure 2(d) backward-edge mapping for more details) is based on a precise forward-edge analysis which determines a set of legitimate calltargets for each function pointer based callsite. The forward mapping is built by matching callsites and calltargets as follows. For each callsite, the number of provided parameters, their type, and the return type is used to build a per function signature. This callsite information is matched with all calltargets in the program during compile time by comparing the number of parameters consumed, their type, and the return type with the data collected at the callsites (not object dispatches). For each matching function signature (calltargets) and callsite signature pair, a set is built. Lastly, for each calltarget contained in the previously determined calltarget set, another per-calltarget return set is built by following the caller-callee function calling convention. This essentially means that for each calltarget-callsite mapping, the legitimate calltarget set contains the return address located after the legitimate callsite.

Class Hierarchy Based (for virtual callees). Forward-edge information, provided by the class hierarchy, is used to map backward edges (see Figure 2(a) forward-edge mapping and Figure 2(b) backward-edge mapping). Essentially, all legitimate forward edges are *reflected* back based on the legitimate class sub-hierarchy these are allowed to call in the first place. As such, for each callee, a set containing all calltarget return sites is built. Depending on the base class of the dispatched object at the callsite and the location in the class hierarchy, the number of calltarget return sites differs. Lastly, our technique is aware of inherited members and as such these are included in the relevant backward-edge target set.

5 DESIGN AND IMPLEMENTATION

In this section, we present first the main components of ρFEM and then we describe how ρFEM handles various function call types including direct calls in §5.1, virtual calls in §5.2, indirect function pointer based calls in §5.3, and how checks are performed at the callees in §5.4. Lastly, implementation details are presented in §5.5.

Figure 4 depicts the components of ρFEM by starting (from left to right) with the callsite checks and performing a walk-through of the involved Clang/LLVM parts. At a high level, ρFEM assigns unique IDs to each function (callee). At each callsite, the ID (or the range of IDs) of the function(s), which can be called by this callsite, is inserted. Before a callee returns, we retrieve the ID data from the

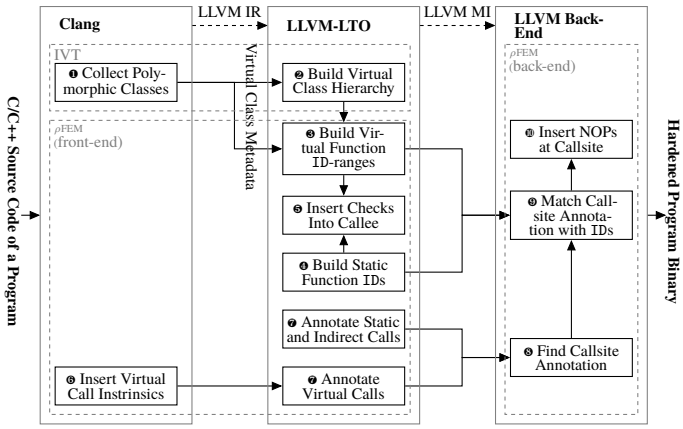

Figure 4: Overview of ρFEM's system design.

callsite, using the return address stored on the stack, and verify the backward edge by comparing the retrieved data with the fixed ID (or ID range) of the callee.

The IDs and callee side checks are generated as follows. The virtual class metadata is collected in Clang ❶ and is used to reconstruct the virtual table hierarchy during the link time optimization (LTO) phase ❷. Afterwards, metadata about the function entries in the virtual tables is used together with the virtual table hierarchy to generate IDs for virtual functions and corresponding ranges for virtual function callsites ❸. Once all virtual functions have been assigned an ID, ρFEM continues to assign IDs to the remaining non-virtual functions ❹. Next, IR code instrumentation based checks are inserted before the return instructions in each function (callee) using the virtual and non-virtual function IDs determined beforehand ❺.

Further, for callsites' ID assignment, ρFEM performs the following steps. In the Clang front-end, each virtual function call is annotated using an LLVM intrinsic ❻. An intrinsic is a particular type of Clang program annotation, which allows marking of certain program parts. This enables inspection at a later phase along the compilation pipeline. During LTO, these intrinsics are detected and the corresponding call instructions are further annotated for later back-end analysis ❼. All remaining calls, *i.e.,* the ones that were not annotated in the front-end, are marked as either direct calls or function pointer-based indirect calls ❼. In the LLVM back-end, all annotated calls are relocated ❽ and then matched with the correct ID or an ID-range depending on the annotation type ❾. That is, virtual callsites are assigned ranges of IDs, direct callsites are assigned the unique ID of the function called directly, and callsites which use function pointers are assigned their respective function signature ID. Next, NOP instructions are inserted directly after the callsite, carrying the ID data as a *payload* ❿. Inserting the NOPs this late in the compiler analysis pipeline (*i.e.,* machine instruction (MI) generation stage) ensures that no instructions are placed between the callsites and our NOPs by a different LLVM pass.

5.1 Handling Direct Calls

Note that CFG forward edges stemming from direct calls do not need to be protected, as the address to which the program control flow is transferred is fixed. Within the context of this paper, a direct call is a forward-edge based program transfer where the target address

is determined by the compiler and available during compile time. Further, we assume that this address is write-protected and cannot be overwritten during runtime.

In contrast, the situation is different for callees which are called through forward direct calls. Since the attacker may manipulate the callee return address on the stack, these backward edges need to be protected. ρFEM handles backward edges returning from direct calls by checking their IDs. Note that each direct function gets its own ID. Direct callsites only have a single target, for which a single valid ID can be determined.

5.2 Handling Virtual Calls

In this subsection, we explain how ρFEM handles virtual function calls. At a high level, ρFEM uses a modified version of the interleaved virtual tables (IVT) metadata as presented by Bounov *et al.* [39]. More specifically, it constructs backward-edge mappings of legitimate return address sets for virtual functions (callees) by first building the virtual table hierarchy and enforcing the respective legitimate virtual table sub-hierarchy for each callee. The matching callees are identified after examining the offset addresses collected within the virtual tables. This is achieved while the matching callers are identified by searching through all virtual callsites. This metadata consists of virtual table hierarchies for all class hierarchies of a program compiled with the Clang/LLVM compiler. Additionally, we modify this metadata layout to have it contain: (1) virtual table entry information, and (2) virtual table offsets which are used during our analysis. ρFEM uses this modified metadata to determine legitimate callsites and to infer forward-edge information.

The Invariant. In order to construct the forward-edge target set for a virtual callsite, we need to be able to represent the IDs of the functions contained in this set as a compact range as this allows a more efficient type of check. Note, that we do not need a total ordering of all function IDs. Instead, we have to be able to build compact (no gaps) ranges. Meanwhile, we note that the type of an object at a particular callsite can either be of precise class type or of any subclass type thereof. Furthermore, if the object has dynamic type of precise class, then, per definition, the callsite uses the implementation provided by the base class. Therefore, any subclass of a precise class can at most call the implementation found in the base class or an override of this implementation. Thus, the sub-graph of the virtual table hierarchy rooted at the base class contains all possible function implementations (*i.e.,* all callees for this callsite). Hence, this sub-graph provides the set of all functions which can be called at this particular callsite, *i.e.,* the target set of the callsite.

Next, ρFEM uses this sub-graph to assign IDs in order to ensure the previously described invariant. There are four steps, which are explained in their order of execution in the following. Note that the algorithm's Step I up to Step III run at LTO, *i.e.,* at compile-time after static linking, while Step IV of the analysis runs in the LLVM back-end, *i.e.,* after LLVM-IR was lowered to machine instructions, and right before the program binary is emitted.

Step I – Constructing Function Hierarchies. Using the modified virtual table hierarchy, a regular class hierarchy is reconstructed, *i.e.,* if a class has multiple virtual tables, these are merged into a single node in the class hierarchy. Figure 5(a) shows such a class hierarchy reconstruction process. Note that each box depicted in Figure 5 represents a virtual table and the first line in each virtual table states the class name from which the virtual table was inherited.

The reconstructed class hierarchy depicted in Figure 5(a) allows ρFEM to collect information about each virtual function implementation. This information is represented as follows. First, the class in which the implementation was defined, and second, whether this class is the root class of the given function. In case it is not the root class, then the implementation has been overridden by a sub-class implementation, *i.e.,* ρFEM is able to analyze all function overrides by using the virtual table metadata.

This information is extracted from the class hierarchy as follows. ρFEM starts by topologically sorting the class hierarchy, which ensures that each parent is visited before any of its children. Then, the topologically-sorted list of classes is traversed and each function entry in the virtual tables of the class is inspected. In case a class contains a function implementation that was not previously encountered, the function is regarded as it would be implemented by this class. This happens, due to the fact that parents are visited before their children during pre-order traversal.

Lastly, to differentiate between root classes and overridden classes (inheriting classes), ρFEM inspects the primary virtual table of the direct parent. In case such a parent class exists and the parents' primary virtual table contains an entry at the same offset as the function in the child, then this child overrides the entry in the parent with a new function implementation. Otherwise, the child defines a new function and becomes the root class for this function. Figure 5(b) shows the root class and the override information inferred within this step.

Step II – Function-wise Traversal and Analysis. The root class information from Step I allows ρFEM to inspect the sub-graph of the virtual table hierarchy rooted at the root class. As such, if ρFEM would disregard virtual inheritance, then this sub-graph would be a tree. By using this information, ρFEM iterates over non-overriding functions and then in an inner loop it iterates through the sub-graph (contains individual virtual tables and not classes) rooted at the root class of the particular function.

Next, ρFEM assigns unique IDs to each virtual table and function pair, while ensuring the following invariant. For a particular function, each parent virtual table has to have a larger ID value than all its children IDs which have a smaller value. This is similar to the well-established heap invariant and is achieved using a pre-order traversal of the corresponding sub-graph. To achieve this, ρFEM starts with the first function, iterates through the sub-graph rooted at the root class of this function, and assigns IDs to each explored virtual table. This process is repeated by ρFEM for the sub-graph of each non-overriding function.

At the same time, the virtual call ID ranges are constructed as follows. For each virtual table, a range of IDs is assigned containing its own ID and the IDs of all of its children. Due to the pre-order traversal, this results in a single closed (compact) range with its own ID having minimum value. Note that these ranges are assigned for each individual function or virtual table pair. In case the sub-graph is a tree, then no virtual inheritance is involved. Consequently, each virtual table gets at most one ID and one range per function contained in the virtual table. Otherwise, the virtual table will get multiple IDs and ranges, since w.r.t. virtual inheritance a virtual table can have multiple parents and therefore can be explored from multiple paths in

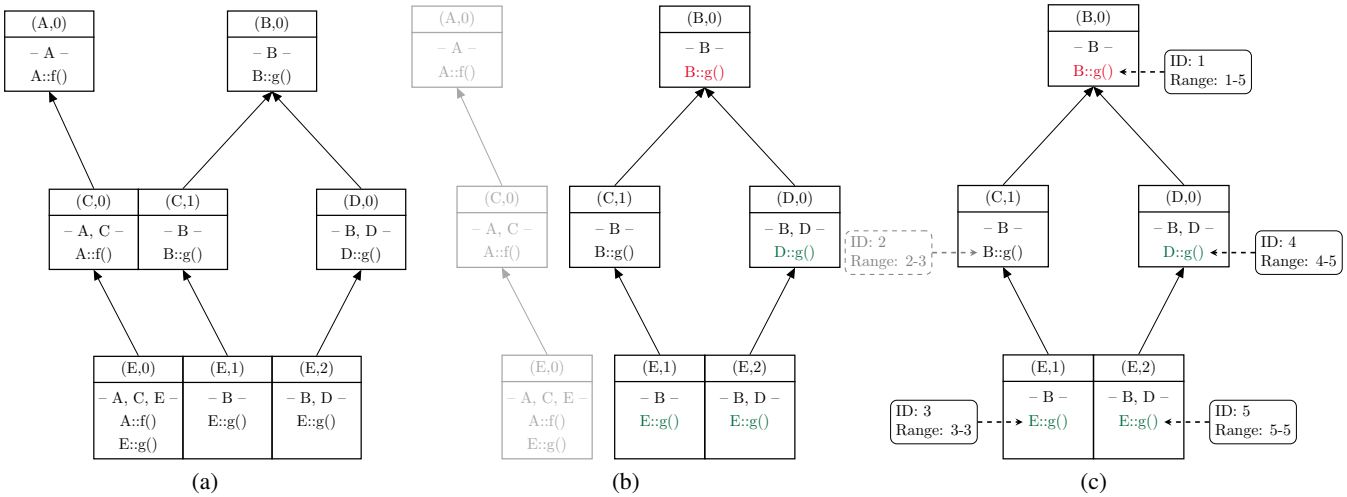

Figure 5: Steps to compute the IDs and the range for function `B::g()` (marked with red font color, upper box in (c)). (a) Step I: Building class hierarchies from virtual table hierarchies. (b) Step II: Collecting root class information of functions (shaded in red font color) and overrides (shaded in green font color). (c) Step III: Determining ranges, calculating their widths and generating IDs for *e.g.,* `B::g()`.

the same sub-graph. Figure 5(c) shows the IDs and ranges assigned for function *g()* [1, 5] with the sub-graph rooted at the virtual table (B,0). Further, in case all classes use non-virtual inheritance, each virtual table gets at most one ID. Note that the three virtual tables shaded in gray in the left part of Figure 5(b) are not part of the sub-graph and therefore have no ID assigned.

Step III – Inserting Callee Backward-edge Checks. In this step, ρFEM inserts checks at each callee. For each virtual function in the program, ρFEM uses the information from Step I to determine the class it belongs to. Then, ρFEM takes each virtual table of this class and looks up the IDs assigned to the virtual table function pair. Note that ρFEM might find multiple IDs, either because there can be multiple virtual tables for a class or because with virtual inheritance there can be multiple IDs for a single virtual table. It is interesting to note that at the end of the analysis the total number of IDs is independent of whether virtual or non-virtual inheritance was used. This is due to the fact that the number of edges in the virtual table hierarchy is independent of the inheritance type. All IDs are unique IDs for this function, because the IDs were assigned for function/virtual table pairs, and a function cannot be defined twice within a virtual table.

Next, ρFEM generates the actual check during LTO. The inserted check works as follows. First, it takes the callee's return address from the stack and second, it tries to load the range data from this address. The `NOP` instructions containing this data are inserted in Step IV. ρFEM then checks whether or not one of the callee IDs is inside of the fetched range. If this is the case, then the check passes. Otherwise, the return address is not an address after a valid callsite for this particular calltarget and program execution is interrupted.

Step IV – Attaching Callsite Metadata. ρFEM annotates, within the front-end, each callsite with its base class and the function implementation of the base class, which legitimately can be called by the callsite. As explained previously, a callsite can only call functions in the sub-graph rooted at the callsite's base class. This principle is reflected by the range, which is obtained through the analysis of base class/function pair. This idea holds, because the range contains only the IDs assigned to the function implementations in the base class or in any of its children classes, which were explored by ρFEM during the pre-order traversal in Step II.

Lastly, ρFEM performs in Step III and also in Step IV the following optimization. As opposed to storing the start and end ID of a range, ρFEM stores the start ID and the width of the range. This optimization reduces the amount of operations required during a runtime check and as such runtime overhead.

5.3 Handling Function Pointer Based Calls

Each non-virtual callee, of which the address escapes to memory (address taken (AT)) can potentially be called by a function pointer based callsite (a non-virtual callsite). The matching callees are identified after removing from the total set of callees the virtual and statically called callees. This is done while the matching callers are identified by removing from the total set of callees the virtual callees and the static address callees. As fully precise control flow analysis is generally impossible due to the fact that alias analysis is undecidable [2], and less precise alias analysis does not necessarily provide small target sets, we assume in this work that it is valid for each function pointer based callsite to call a callee, as long as the function signatures match. As such, ρFEM implements a function signature encoding, allowing it to encode function signature data in IDs. The function signature computed by ρFEM consists of: (1) the name of the caller function and of the callee function, (2) the number of parameters the caller provides and the number of parameters the callee consumes (as for now, the first 8 function parameters are taken into consideration), (3) their parameter types (as for now, 26 LLVM IR parameter types, *i.e.,* `HalfTyID`, `FloatTyID`, `VoidTyID`, *etc.* are taken into consideration), and (4) the callee return type.

Consequently, for an address taken (AT) function, ρFEM generates a function signature ID by using the previously mentioned function signature encoding algorithm. Using the same encoding, ρFEM annotates each function pointer based callsite with such an ID. The callee accepts both the ID(s) generated in §5.1 or §5.2 alongside

with the function signature ID in case it was called indirectly, but only if its own function signature was used at the callsite. Note that the regular ID is the ID which was assigned through the pre-order traversal. Lastly, during runtime, in case the caller signature matches the callee signature, the control flow is allowed to return. Otherwise, the control flow transfer is stopped.

5.4 Callees Integrity Checks

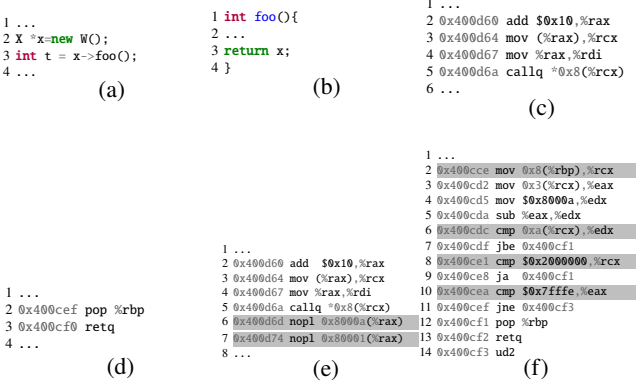

Figure 6: Caller (a), callee (b) C++ code; caller (c) machine code (no instrumentation). Callee (d) assembly (no instrumentation); caller (e), callee (f) instrumented machine code.

Figure 6 depicts the instrumentation added by ρFEM to a caller and its corresponding callee in order to protect against backward-edge control flow violations. For generating the values used in Figure 6(e) lines 6 and 7, we use a counter.

Range based checks. The code listings depicted in Figure 6(a) and Figure 6(b) show the original source code, while the code listings depicted in Figure 6(c) and Figure 6(d) show the resulting assembly instructions without applying any backward-edge checks. Lines 2 – 4 in Figure 6(c) execute the virtual dispatch using an object *X* stored in the `rax` register. Before the callee returns, as depicted in Figure 6(d), the stack is popped once to clean up the stack frame.

Further, we analyze the actual checks shown in the last row contained in Figure 6(e). The newly-added `NOP` instructions are depicted on lines six and seven in Figure 6(e) and contain a range starting at `0x0a` (`StartID`) with a width of `0x01` (`WidthOfRange`), *i.e.,* the only callees valid at this callsite have IDs `0x0a` or `0x0b`. When looking at the callee's instructions depicted in Figure 6(f), we observe that its ID is located on line 4. As expected, it has one of the IDs inside the range, namely `0x0a`. We can also see the range in the two instructions before (lines two and three): the start ID is loaded from the first `NOP` and then the callee ID is substracted from it (`StartID−0x0a`). In case everything went through up to this point, the result of the subtraction should now be in the range from 0 to `WidthOfRange`, which is checked with the help of the `cmp` and `jbe` instructions located on lines six and seven in Figure 6(f).

Signature-based checks. Similarly, indirect calls (*i.e.,* function pointer based), which have a matching function signature encoding (*e.g.,* the `cmp` with the address `0x7fffe`; see line 10 in Figure 6(f)), which passes the check then the execution continues. This value represents the valid function signature encoded as a hash value. The hash value is a word size value obtained by concatenating the number of parameters, their types and return type as a string and then hashing them as described by vTrust [40].

Error handling. In case none of the checks succeed, the program executes the `ud2` instruction depicted on line 14 in Figure 6(f), causing the program to terminate. While this type of mitigation is sufficient for our purposes, in real-world applications more sophisticated error handling might be used. Instead of abruptly terminating the program, another possible approach is to log each legal and illegal backward-edge transfer.

External Calls. In case a protected function is called by an unlabeled callsite (*i.e.,* external library call), then this call causes the protected function to return at the next address after the call instruction with the help of the instructions located on line 7 and 8 in Figure 6(f). ρFEM is able to differentiate between external and internal calls by determining during compile time the address range of the hardened program. As such, external calls have a memory address not contained in the range of the protected program and thus the inserted check can differentiate between internal and external calls.

5.5 Implementation Details

We implemented ρFEM as five Clang/LLVM analysis passes, as follows. One front-end pass for collecting metadata from the Clang compiler for later usage during LTO analysis, three LTO passes and a machine instruction-level pass. For this purpose, we extended the Clang/LLVM [26] compiler framework infrastructure. As three of the four ρFEM passes are performed during link time, our system requires LLVM's LTO. As previously mentioned, the implementation of ρFEM is split between the Clang compiler front-end (metadata collection), three new link-time passes and one machine-level pass used for analysis and generating backward-edge constraints, totaling around 3,235 LOC. ρFEM supports separate compilation by relying on the LTO mechanism built in LLVM [26]. ρFEM generates unique IDs by keeping track of already assigned ones and continuously incrementing a counter variable for generating new IDs. Lastly, by carefully traversing each class hierarchy in pre-order, unique ID assignment is guaranteed.

6 EVALUATION

In this section, we address the following research questions (RQs).

- **RQ1:** How *effective* is ρFEM in protecting backward edges (§6.1)?
- **RQ2:** What backward-edge *attacks* can ρFEM thwart (§6.2)?
- **RQ3:** What *security benefit* does ρFEM offer (§6.3)?
- **RQ4:** What is the *runtime overhead* of ρFEM (§6.4)?

Benchmark Programs. In our evaluation, we used the following real-world C/C++ programs: (1) Memcached [41] (general-purpose distributed memory caching system, v.1.5.3, C/C++ code), (2) Nginx [42] (Web server, usable also as: reverse proxy, load balancer, mail proxy and HTTP cache, v.1.13.7, C code), (3) Lighttpd [43] (Web server optimized for speed-critical environments, v.1.4.48, C code), (4) Redis [44] (in-memory database with in-memory key-value store, v.4.0.2, C code), (5) Apache Httpd Server (Httpd) [45] (cross-platform Web server, v.2.4.29, C code) and the following C++ programs: (6) NodeJS [46] (cross-platform JS run-time environment, v.8.9.1, C/C++ code), (7) Apache Traffic Server [47] (modular, high-performance reverse proxy and forward proxy server, v.2.4.29, C/C++ code), and

(8) Google Chrome [48] (Web browser, v.33, C/C++ code). These programs were selected due to their real-world security relevance.

Experimental Setup. The benchmarks were performed on an Intel i5-3470 CPU with 8GB RAM running on the Linux Mint 18.3 OS. We carefully compiled each program and executed it ten times in order to provide reliable mean values. Note that we re-applied for all hardened programs, in case these existed, their functionality/correctness tests and we can confirm that all work as expected after hardening with ρFEM. Lastly, all programs were compiled with Clang/LLVM -O2 compiler optimization flag.

6.1 Protection Effectiveness

In this section, we assess the precision of ρFEM by counting the number of allowed return targets per function (callee). Further, we consider the size of the return target set an indicator for the precision of ρFEM's backward-edge protection. Note that for all tables in this section, we use the following measures: #Callees (all callees), minimum, 90th percentile, maximum, geomean, median, average, and standard deviation.

Program	#Callee	Min	90p	Max	Geo	Med	St.dev
Httpd	1,086	0	20	187	3.34	3	18.18
Lighttpd	451	0	6	317	1.97	1	19.46
Memcached	106	0	8.5	136	2.81	2	15.15
Nginx	1,132	0	29	1,630	3.29	2	58.23
Redis	2,644	0	7	3,796	1.97	1	81.74
NodeJS	30,330	0	231	6,837	3.34	1	114.11
Tr. Server	6,115	0	14	2,673	3.13	2	64.04
geomean	1,616.32	0	18.23	986.87	2.77	1.57	40.74

Table 1: Return addresses for virtual and non-virtual functions.

Table 1 depicts the number of functions hardened by ρFEM and the size of their legitimate return address sets enforced by ρFEM. The geometric mean value for all assessed programs is 2.77 return addresses per callee. This considerably decreases the chances of a successful attack. The average value obtained for NodeJS (most complex program analyzed in Table 1) represents an outlier. This value originates from the high number of small helper functions which are not in-lined and a large number of function pointer based indirect callsites. We further investigated the results for NodeJS and observed many indirect callsites calling template functions, which were generated for multiple JavaScript types. These functions have the same signature and can therefore be targeted by many indirect callsites. Further, we note that for callees which allow more than 10 return targets, depending on the gadget types, there is potentially a considerable decrease in the provided protection level.

Program	#Callee	90p	Max	Geo	Med	St.dev.
NodeJS	4,177	239	2,792	23.12	19	143.31
Tr. Server	948	25	992	7.54	11	59.8
Chrome	66,032	1,150	15,014	144.21	155	3,677.36
geomean	6,394.53	190.11	3,464.50	29.29	31.87	315.86

Table 2: Allowed return addresses for only virtual callees.

Table 2 depicts the sizes of the legitimate return target sets for only virtual functions. By comparing the geomean results depicted in Table 2 with the results shown in Table 1 (2.77 vs. 29.29) we note that in general ρFEM performs better for non-virtual functions than for virtual functions. This is due to the fact that ρFEM: (1) uses ranges for virtual functions instead of single IDs (*i.e.,* ranges contain more than one element, since the class sub-hierarchy is enforced backwards), and (2) cannot precisely determine when a virtual function is called through a function pointer based call (due to the currently used address taken analysis). Further, note that in general it is difficult to rule out pointers to virtual functions because pointers to these functions are already stored in the corresponding virtual tables. Therefore, we opted for the most conservative implementation in ρFEM in which we assume that most of the virtual functions can be called using function pointers which are indirect callsites as well. For this reason, we have in general more legitimate calltargets for these callees.

Program	Base	Min ‰	90p ‰	Max ‰	Geo ‰	Med ‰	SD ‰
Lighttpd	52,060	0	0.38	3.59	0.06	0.06	0.35
Memcached	24,672	0	0.34	5.51	0.11	0.08	0.61
Nginx	173,273	0	0.17	9.41	0.02	0.01	0.34
Redis	333,835	0	0.02	11.37	0.01	0.00	0.24
NodeJS	2,479,736	0	0.09	2.76	0.00	0.00	0.05
geomean	179,091.65	0	0.13	5.67	0.02	0.01	0.24

Table 3: Fraction of instructions per mil. allowed to return to.

Table 3 depicts the fractions of instructions a callee can target. The results denote the fraction (in ‰) of return targets allowed per callee. The *Baseline* entry denotes the number of assembly instructions (addresses) in the program binary code section(s). We used the objdump tool to determine the *Baseline* entries. Note that without any backward-edge protection a return instruction can freely transfer control flow to any of the *Baseline* addresses. The results depicted in Table 3 are important since these show the fraction of legitimate addresses which are allowed to be called after we hardened the binary with ρFEM. The results in the second column up to the last (from left to right) were determined by dividing the results from Table 1 with the total number of *Baseline* instructions depicted in column 2 (*Baseline*) of Table 3. The results indicate that the fraction of addresses, which are targetable after applying ρFEM for every analyzed program, is less than one in a thousand addresses on average. Thus, in geomean less than one address can be targeted by more than 10K addresses when considering the whole program.

6.2 Exploits Defended

In this section, we show ρFEM's exploit coverage by creating a suite of C/C++ programs in order to demonstrate various possible scenarios of calltarget return address overwrite prevention. These programs are based on five ROP primitives (see T1-T5 in Table 4) identified and confirmed [49] to be representative for ROP exploits.

Exploit	Stopped	Remark
Active-Set Attacks [49, 50]:		
Type 1 (**T1**)	✓	Return to any stack func.
Type 2 (**T2**)	✓	Return to a child process
Type 3 (**T3**)	✓	Return to earlier callsites
Type 4 (**T4**)	✓	Return to future callsites
Type 5 (**T5**)	✓	Return to program begin
CALL-ret violating [51]:		
Innocent flesh on the bone	✓	Caller-callee function calling conv. violation

Table 4: Stopped backward-edge attack types.

Table 4 presents a summary of several types of backward-edge based attacks and the primitives on which these rely. For each of the types from **T1** up to **T5**, our suite contains at least one program reflecting this behavior. **T1.** Return to any active function on the stack (not just the last function put on the call stack). **T2.** Return to parent code in a child process after a fork. **T3.** Return to earlier callsites in functions on the stack. **T4.** Return directly to future callsites in functions on the stack. **T5.** Return directly to the beginning of a program (typically the second callsite in main). Next, we present the backward-edge primitives on which the attacks depicted in Table 4 rely. Note that **T1** up to **T5** can independently be used to bypass the following backward-edge protection [49, 50, 52] techniques.

Type	Full CFL	ρFEM
T1	✗	✓
T2	✗	✓
T3	✗	✓
T4	✗	✓
T5	✗	✓

Table 5: Detected exploit types, T1-T5.

Table 5 depicts the results after running the programs from our test suite with ρFEM. Note that we could not evaluate these with the CFL tool as this is not open source. However, based on the analysis of the CFL [5] paper, we expect that CFL cannot detect any of these 5 types of backward-edge violations as it allows a callee to return to any address following an indirect or direct function call. Further, we explain how ρFEM mitigates these attacks. **T1**: none of the addresses enforced by ρFEM is located at a function start, only legitimate function return addresses are allowed. **T2**: in case the return address is not in the legal return target set of the particular return, then this is forbidden. **T3**: callsite addresses are rejected by ρFEM completely, allowed addresses are only these which are following a callsite. **T4**: future callsite addresses are not included by ρFEM in the target address set and as such these are forbidden. **T5**: callsite addresses are completely forbidden by ρFEM. Lastly, ρFEM stops the Galileo ROP attack [51]. This covers returning to arbitrary code in mapped libraries as well. This is due to the fact that ρFEM forbids that the callee can return to any program address.

6.3 Security Analysis

In this section, we evaluate the availability of gadgets after hardening the program with ρFEM. Assuming that the initial backward edge is protected by ρFEM, three conditions have to be met to make a gadget usable in a ROP chain: (1) the gadget has to start with a NOP instruction (in order to be targetable from a secured backward edge), (2) the payload of the NOP instruction has to pass the backward-edge check of the incoming backward edge, and (3) the return instruction at the end of the gadget has to be either unprotected or its target has to be contained in the return target set of the function the gadget is part of. Note that condition (2) has already been extensively discussed in RQ1 in a generalized form. Assuming the return target set of the gadget is not sufficient to extend the chain to the next gadget (as shown in RQ1), then condition (3) only holds if the backward edge is unprotected.

Table 6 depicts the results obtained after analyzing the hardened and unhardened program binaries using the ROPgadget [53] tool

Program	# LTO	# ρFEM	# no-NOP	# ret check	# not prot.	% not prot.
Httpd	12,664	19,723	19,430	11,033	77	0.39%
Lighttpd	5,855	7,309	7,154	1,100	132	1.81%
Nginx	15,789	20,392	20,212	8,475	128	0.63%
Memcached	1,805	2,056	2,007	184	43	2.09%
NodeJS	375,032	490,570	485,396	99,853	3,222	0.66%
Tr. Server	75,187	106,766	104,935	23,486	1,275	1.19%
geomean						0.95%

Table 6: Number of `ret` gadgets before and after hardening.

(ROP gadget finding tool). We passed the following arguments to ROPgadget: `-depth=30 -nosys -nojop`.

Next, we consider the Table 6 columns numbered from left to right; in total, we have seven columns. The second column (**# LTO**) in Table 6 shows the number of unique gadgets found in the unhardened binary with only LTO enabled. We consider a gadget to be unique if it consists of a unique sequence of instructions.

The third column (**# ρFEM**) presents the number of unique gadgets in the hardened binaries with ρFEM. In this case, for a gadget to be unique, we also consider the existence of a `NOP` instruction at the beginning of the gadget. This means that two gadgets with the same sequence of instructions, one with `NOP` before these instructions and one without, are different gadgets. This differentiation is important since for an attack the existence of the leading `NOP` is relevant. Due to this change, the number of gadgets found by ROPgadget [53] increases compared to the unhardened binary.

The fourth column (**# no-NOP**) shows the number of gadgets in the hardened binary that fail condition (1), *i.e.,* these gadgets do not start with a `NOP` instruction and are therefore not targetable by a secured backward edge. We can observe that most gadgets do not start with a `NOP` instruction as expected, since the number of callsites (and therefore of `NOP` instructions) is small compared to the total number of instructions.

The gadgets depicted in column five (**# ret check**) end with a return check generated by ρFEM and therefore can only target gadgets that match the return target set of the function the gadget is contained in. Note that the return instructions not protected by ρFEM are mostly either boilerplate functions (*e.g.,* global setup and init functions) or small stack adjustment functions generated by LLVM. These functions rarely contain any calculating instructions (*e.g.,* `ADD`, `SUB`) with non-immediate operands.

The sixth column (**# not prot.**) shows the number of gadgets for which both conditions (1) and (2) hold. These gadgets are not counted in column two or three and therefore have a `NOP` instruction at the beginning, but no check of the backward edge. These are the remaining gadgets that can be used for a ROP chain attack without exploitation of the residual return target set evaluated in RQ1.

Lastly, the seventh column (**% not prot.**) depicts the percentage of all gadgets in the hardened binary that are not protected after hardening the program binary with ρFEM. We can observe that in geomean only 0.05% (1 - 0.95%, see Table 6) of the previously found gadgets are still useful after hardening the program binary with ρFEM. Thus, ρFEM protects in geomean > 99% of the identified gadgets from usage during an attack without averaging the results.

6.4 Runtime Overhead

In this section, we evaluate the runtime overhead of ρFEM by using the SPEC CPU 2017 benchmarks and real-world programs.

Benchmark	LTO	SafeStack + LTO	SafeStack + LTO %	ρFEM + LTO	ρFEM + LTO %
500.perlbench_r	11.0	11.1	-0.91	11.1	-0.91%
505.mcf_r	11.7	11.4	2.56%	11.7	0.00%
520.omnetpp_r	8.05	8.12	-0.87%	8.02	0.37%
523.xalancbmk_r	8.79	8.79	0.00%	8.61	2.05%
525.x264_r	18.6	19.4	-4.30%	18.5	0.54%
531.deepsjeng_r	13.6	14.0	-2.94%	13.4	1.47%
541.leela_r	12.7	12.8	-0.79%	12.6	0.79%
557.xz_r	8.73	9.13	-4.58%	8.74	-0.11%
508.namd_r	12.8	13.0	-1.56%	12.7	0.78%
511.povray_r	17.7	17.6	0.56%	16.9	4.52%
519.lbm_r	4.82	4.82	0.00%	4.82	0.00%
526.blender_r	17.1	17.0	0.58%	17.1	0.00%
538.imagick_r	17.6	19.1	-8.52%	17.7	-0.57%
600.perlbench_s	3.51	3.46	1.42 %	3.55	-1.14 %
605.mcf_s	6.69	6.31	5.68 %	6.67	0.30 %
620.omnetpp_s	3.81	3.7	2.89 %	3.66	3.94 %
623.xalancbmk_s	3.73	3.65	2.14 %	3.55	4.83 %
625.x264_s	5.15	5.2	-0.97 %	5.09	1.17 %
631.deepsjeng_s	4.06	4.02	0.99 %	4	1.48 %
641.leela_s	3.62	3.54	2.21 %	3.62	0.00 %
657.xz_s	2.06	2.24	-8.74 %	2.24	-8.74 %
geomean	-	-	0.85%	-	0.11%

Table 7: ρFEM's overhead w.r.t. the SPEC CPU2017.

Table 7 depicts the runtime overhead when running ρFEM on all pure `C/C++` programs contained in the SPEC CPU2017 (rate and speed) benchmark. Note that we could not compile `602.gcc_s` and `502.gcc_r` with SafeStack, whereas it was possible with ρFEM. For this reason, we did not include these programs in Table 7. As reported in the last row of the table, the average runtime overhead of ρFEM is 0.11% comparing favorably to SafeStack's overhead which is 0.85% on average. Thus, these results support ρFEM's competitiveness.

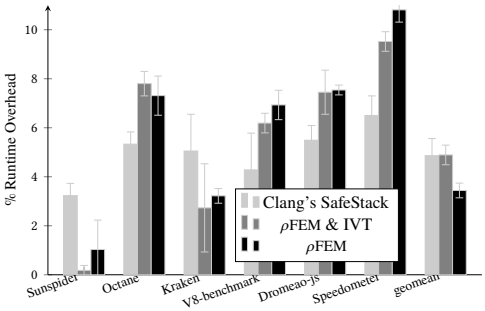

Figure 7: Comparing ρFEM, SafeStack and IVT overheads.

In addition to the SPEC CPU 2017 benchmark, we also evaluated ρFEM with a set of popular JavaScript benchmarks for the Chrome Web browser. Figure 7 depicts the runtime overhead of SafeStack (shaded light gray), ρFEM + IVT (shaded gray; note IVT provides virtual call based forward-edge protection only), and ρFEM (shaded black) for these benchmarks. As shown in the figure, when running ρFEM incrementally together with IVT, offering both forward-edge and backward-edge protection, the geomean runtime overhead is equal to that of SafeStack (4.86% vs. 4.89%). In contrast, when running ρFEM alone then the runtime ovehead is 3.44% in geomean. Therefore, we conclude that ρFEM's runtime overhead is negligible and that it can be used as an always-on solution.

7 LIMITATIONS AND FUTURE WORK

Number of all function returns. Clang SafeStack's (based on LLVM v.3.7.0 stable) number of return addresses per callee is smaller than the number of return address which ρFEM can enforce. In future work, we want to reduce the number of return addresses per callee. To achieve this, we first need to check availability of gadgets for each hardened program. Additionally, we will reduce this number by performing an analysis of provided and consumed parameters by each callsite. This will further help to reduce the number of callee return addresses considerably.

Attacker access to legitimate gadgets. As the number of callee return addresses is in general larger than one, in some situations, these addresses could still be used by an attacker to jump to a legitimate address, in order to access useful gadgets. In future work, we would like to address this issue by analyzing all legitimate callee return sites and determining if these can be used as a gadget. In case this can be further used as a gadget then instruction level program transformations will be made in order to make the gadget unusable.

Performing attacks with our protection in-place. The legitimate number of return addresses for a callee protected by ρFEM is low but research shows that in general attacks are still feasible. For example, Carlini *et al.* [28] show that in case the attacker knows: (1) the legitimate return address of a callee, and (2) a usable return address to access a gadget, then an attack is still possible. In future work, we plan to address these cases by first looking into possibilities to insert another level of indirection (*e.g.,* re-purposed register based trampoline) (1) between callee and its return address and (2) between callee and the return address which can be used to jump to the beginning of a gadget.

`RET` instrumentation improvement. The return address is used to access the respective return site(s), and to check IDs. This address will be reloaded from the stack, via a `ret` instruction, and eventually consumed. As such, this scheme may suffer from TOCTTOU-like issues, as the check can be completed successfully, but the return address in the stack can be altered right before `ret` is executed. In future work, we will address this issue by replacing `ret` with `pop` and `jmp` instructions, such that the return address is not double-fetched. Note that most of backward-edge CFI schemes do not do this (*i.e.,* are willing to tolerate the risk of TOCTTOU) because replacing RET with `POP/JMP` may greatly increase overhead. This was the case at least on Intel CPUs a few years ago as this would greatly influence the hardware return address prediction.

Tail calls and position independent code. Currently, ρFEM does not support tail calls and position independent code (PIC). In future work, we plan to address this issue by keeping track during runtime of all function calls which have not returned and enforce that a tail call could return to the next address of all functions which did not return. The PIC issue can be also addressed by not using absolute addresses and by compiling any PIC code that may be loaded in protected programs.

Labeling of legitimate return sites. ρFEM inserts labels with `IDs` that correspond to each legitimate return site for each function return (callee). This could provide an attacker a useful hint at which addresses it is legitimate to return. In case these return addresses contain

useful gadgets, then the attacker may return at these addresses, thus bypassing the ρFEM CFI checks. This limitation can be addressed in future work by adding an additional level of indirection and computing the ID, which was previously inserted at the `NOP`s location, on the fly.

Control flow bending. Control Flow Bending (CFB) [28] showed that, even with fully-precise static CFI, powerful CRAs are still possible. ρFEM cannot handle CFB attacks with the same precision as shadow stack techniques. Note that all other techniques (excluding shadow stacks) cannot protect against CFB as well. Further, to date only shadow stack can mitigate this type of attack, but shadow stacks can be bypassed [19]. Thus, in future work, we plan to make ρFEM able to mitigate CFB attacks as well, at least to a partial degree.

Inter-modular support. ρFEM can only secure single binaries. Thus, each used dynamic library has to be compiled in order to be protected. In the current ρFEM implementation, the `ID`s for different modules may overlap, which increases the return target set. In addition, inter-modular backward edges are not protected. In future work, we would like to address this by synchronizing `ID`s between modules. As a result, the program modules are compiled after compiling dynamic libraries. This allows for forward sharing of `ID` information in the modules, which use the dynamic libraries. As such, we consider this as an engineering limitation that is easily solvable.

Imprecise function pointer callsite analysis. Our experiments show that function pointer based callsites account for a significant amount of return targets. This is especially problematic for virtual callees since these are usually not targets of function pointer based callsites. In future work, this issue can be addressed by developing a better address taken (AT) function detection analysis, which would help to reduce the number of functions that can be targeted by any function pointer based callsite. Lastly, the function signature encoding can be improved by using more data types and the object `this` pointer.

8 RELATED WORK

There are many defenses to protect backward-edges. In the following, we briefly categorize them to differentiate and motivate our research.

Shadow stack based. SafeStack [8] is an LLVM/Clang compiler framework based tool which can protect program backward edges. SafeStack uses for each program function stack a secondary shadow stack frame that will be loaded during runtime. However, Goktas *et al.* [19] show how SafeStack can be bypassed with relatively low effort, thus bypassing this protection technique. PittyPat [4] introduces a fine-grained path-sensitive CFI for protecting both forward and backward edges. It uses shadow stack to maintain a stack of points-to information during its analysis, it will always allow only a single transfer target for each return instruction.

Double shadow stack. Zieris *et al.* [11] propose a leak-resilient dual stack approach by relocating potentially unsafe objects on a second stack while keeping the unsafe objects on the program's original stack. Compared to ρFEM, our approach does not hide information, and does not position data at statically defined addresses as our labels are randomized during each program compilation.

HW register based. Shadesmar [12] is a compact shadow stack implementation that relies on information hiding and which re-purposes two new Intel x86 extensions: memory protection (MPX), and page table control (MPK). It uses a register in order to hide the shadow stack pointer and thus the access to the shadow stack which will be located at variable distances in memory. As such it still relies on information hiding but raises the bar for the attacker when searching for the shadow stack. The authors admit that information hiding is fundamentally broken, but recommend it only because of the resistance to deploying any protection mechanism with greater than 5% overhead.

Non-shadow stack based. CFL [54] is a GCC compiler based tool used for protecting backward edges only by instrumenting the source code (only for 32-bit) of a compiled program. CFL uses a statically precomputed program control flow graph (CFG). Thus, this technique (as it can be observed) relies on the precision of the computed CFG. CFL can protect against code reuse attacks for statically linked 32-bit binaries, which violate the statically precomputed CFG. CFL provides three modes of operation: (1) just alignment, (2) single-bit CFL, and (3) full CFL, which each have different performance overheads. πCFI [55] is a compiler-based tool, which lazily builds a CFG on the fly during runtime. Indirect edges are added in the CFG before indirect branches need those edges. πCFI disallows adding edges, which are not present in the statically computed all-input CFG (this CFG serves as an upper bound for the runtime constructed CFG).

Re-purposed registers. μRAI [3] protects backward edges also without shadow stacks in microcontroller-based systems (MCUS) by using a specific register to memorize where the legitimate return address resides. μRAI's technique relies on moving return addresses from writable memory to readable and executable memory. It repurposes a single general purpose register that is never spilled, and uses it to resolve the correct return location. At runtime, μRAI provides each function a uniquely encoded ID (*e.g.,* a hash value) each time the function is executed.

9 CONCLUSION

We have presented ρFEM, a Clang/LLVM-based backward-edge runtime protection tool, which leverages static forward-edge information of `C/C++` programs to protect backward edges. We conducted an evaluation of ρFEM with several real-world programs such as Google's Chrome Web browser, NodeJS, and Nginx. Our evaluation results show that only a low median number of return targets per callee return site are allowed. More precisely, the median geomean number of return addresses per callee is 1.57 while the geomean is 2.77. These results confirm that the attack surface is drastically diminished, thus the chances of successfully performing a control-flow hijacking attack are considerably reduced. Further, our experiments with Google's Chrome Web browser indicate that ρFEM imposes a low runtime overhead of 3.44% in geomean. Lastly, ρFEM is compatible with currently available real-world `C/C++` applications such as Google's Chrome Web browser, readily deployable, and advances the state-of-the-art protection of program callees.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. Further, we want to thank Thurston Dang (MIT, USA), Artur Janc (Google, CH), Jaroslav Sevcik (Google, DE), and Haohuang Wen (OSU, USA), for constructive feedback on an earlier version of this paper, which helped to improve the quality of our work.

REFERENCES

[1] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[2] G. Ramalingam. The Undecidability of Aliasing. In *Transactions on Programming Languages and Systems (TOPLAS), ACM*, 1994.

[3] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer. μRAI: Securing Embedded Systems with Return Address Integrity. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[4] R. Ding, C. Qian, C. Song, W. Harris, T. Kim, and W. Lee. Efficient Protection of Path-Sensitive Control Security. In *USENIX Security Symposium (USENIX Security)*, 2017.

[5] T. Bletsch, X. Jiang, and V. Freeh. Mitigating Code-reuse Attacks with Control-flow Locking. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[6] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[7] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[8] Clang/LLVM. Clang's SafeStack. https://clang.llvm.org/docs/SafeStack.html.

[9] T. Chiueh and F.H. Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[10] xLab. Return Flow Guard. http://xlab.tencent.com/en/2016/11/02/return-flow-guard/.

[11] P. Zieris and J. Horsch. A Leak-Resilient Dual Stack Scheme for Backward-Edge Control-Flow Integrity. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2018.

[12] N. Burow, X. Zhang, and M. Payer. SoK: Shining Light on Shadow Stacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[13] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium (USENIX Security)*, 2013.

[14] GCC. GCC's Shadow Stack Proposal. 2019. https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-stack_005fprotect-function-attribute.

[15] T. H. Y. Dang, P. Maniatis, and D. Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *ACM Asia Conference on Computer & Communications Security (AsiaCCS)*, 2015.

[16] E. Goktas, B. Kollenda, P. Koppe, G. Bosman, Portokalidis, T. Holz, H. Bos, and C. Giuffrida. Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure. In *European Symposium on Security and Privacy (EuroS&P)*, 2018.

[17] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking Holes in Information Hiding. In *USENIX Security Symposium (USENIX Security)*, 2018.

[18] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, and M. Franz. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[19] E. K. Goktas, A. Oikonomopoulos, R. Gawlik, B. Kollenda, I. Athanasopoulos, C. Giuffrida, G. Portokalidis, and H. J. Bos. Bypassing Clang's SafeStack for Fun and Profit. In *Black Hat Europe*, November 2016.

[20] B. Niu and G. Tan. Modular Control-Flow Integrity. In *Programming Language Design and Implementation (PLDI)*, 2014.

[21] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM. In *USENIX Security Symposium (USENIX Security)*, 2014.

[22] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[23] B. Niu and G. Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[24] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, and H. Bos. ShrinkWrap: VTable Protection Without Loose Ends. In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[25] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[26] Clang/LLVM. Clang/llvm compiler framework. https://clang.llvm.org/.

[27] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *European Symposium on Security and Privacy (EuroS&P)*, 2016.

[28] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium (USENIX Security)*, 2015.

[29] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert. Analyzing Control Flow Integrity with LLVM-CFI. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[30] Industry Coalition. Itanium C++ ABI. https://mentorembedded.github.io/cxx-abi/abi.html.

[31] J. Gray. C++: Under the Hood. 1994. http://www.openrce.org/articles/files/jangrayhood.pdf.

[32] ARM. C++ ABI for the ARM Architecture. 2015. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041e/IHI0041Ecppabi.pdf.

[33] Intel. Intel Control-Flow Enforcement Technology (CET). https://software.intel.com/en-us/blogs/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks.

[34] V. van der Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitivce CFI. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[35] F. Schuster, T. Tendyck, J. Pewny, A. Tendyck, M. Steegmanns, M. Contag, and T. Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[36] Microsoft. The STRIDE Threat Model, 2009. https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN.

[37] P. Muntean, S. Wuerl, J. Grossklags, and C. Eckert. CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM. In *European Symposium on Research in Computer Security (ESORICS)*, 2018.

[38] Microsoft. Microsft's data execution prevention. 2018. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx.

[39] D. Bounov, R. G. Kici, and S. Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[40] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. vTrust: Regaining Trust on Virtual Calls. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2016.

[41] Memcached. Memcached. 2017. https://memcached.org/.

[42] Nginx. Nginx. 2017. https://nginx.org/en/.

[43] LightHTTPD. LightHTTPD. 2017. https://www.lighttpd.net/.

[44] Redis. Redis. 2017. https://redis.io/.

[45] Apache Software Foundation. Apache Httpd. 2017. https://httpd.apache.org/.

[46] Node.js Foundation. NodeJS. 2017. https://nodejs.org/en/.

[47] Apache Software Foundation. Apache Traffic Server. 2017. http://trafficserver.apache.org/.

[48] Google. Google's Chrome Web browser. 2017. https://www.chromium.org/.

[49] M. Theodorides and D. Wagner. Breaking Active-Set Backward-Edge CFI. In *Hardware Oriented Security and Trust (HOST)*.

[50] M. Theodorides. Breaking Active-Set Backward-Edge CFI. In *Technical Report No. UCB/EECS-2017-78*, 2017. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-78.html.

[51] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (On the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[52] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. HAFIX: Hardware-Assisted Flow Integrity Extension. In *Annual Design Automation Conference (DAC)*, 2015.

[53] J. Salwan. ROPgadget. 2018. https://github.com/JonathanSalwan/ROPgadget.

[54] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ACM Asia Conference on Computer & Communications Security (AsiaCCS)*, 2011.

[55] B. Niu and G. Tan. Per-Input Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.