A Tough `call`: Mitigating Advanced Code-Reuse Attacks At The Binary Level

Victor van der Veen*†, Enes Göktaş*†, Moritz Contag‡, Andre Pawlowski‡, Xi Chen†, Sanjay Rawat†, Herbert Bos†, Thorsten Holz‡, Elias Athanasopoulos†, and Cristiano Giuffrida†
*Equal contribution joint first authors

†*Computer Science Institute*
Vrije Universiteit Amsterdam, The Netherlands
{*vvdveen, herbertb, giuffrida*}*@cs.vu.nl,*
{*e.goktas, x.chen, s.rawat, i.a.athanasopoulos*}*@vu.nl*

‡*Horst Görtz Institut for IT-Security (HGI)*
Ruhr-Universität Bochum, Germany
{*moritz.contag, andre.pawlowski, thorsten.holz*}*@rub.de*

***Abstract*—Current binary-level Control-Flow Integrity (CFI) techniques are weak in determining the set of valid targets for indirect control flow transfers on the forward edge. In particular, the lack of source code forces existing techniques to resort to a conservative address-taken policy that over-approximates this set. In contrast, source-level solutions can accurately infer the targets of indirect callsites and thus detect malicious control-flow transfers more precisely. Given that source code is not always available, however, offering similar quality of protection at the binary level is important, but, unquestionably, more challenging than ever: recent work demonstrates powerful attacks, such as *Counterfeit Object-oriented Programming* (COOP), which made the community believe that protecting software against control-flow diversion attacks at the binary level is impossible.**

In this paper, we propose binary-level analysis techniques to significantly reduce the number of possible targets for indirect callsites. More specifically, we reconstruct a conservative approximation of target function prototypes by means of use-def analysis at possible callees. We then couple this with liveness analysis at each indirect callsite to derive a many-to-many relationship between callsites and target callees with a much higher precision compared to prior binary-level solutions.

Experimental results on popular server programs and on SPEC CPU2006 show that *TypeArmor*, a prototype implementation of our approach, is efficient—with a runtime overhead of less than 3%. Furthermore, we evaluate to what extent *TypeArmor* can mitigate COOP and other advanced attacks and show that our approach can significantly reduce the number of targets on the forward edge. Moreover, we show that *TypeArmor* breaks published COOP exploits, providing concrete evidence that strict binary-level CFI can still mitigate advanced attacks, despite the absence of source information or C++ semantics.

I. Introduction

Control-Flow Integrity (CFI) [7] is one of the most promising ways to stop advanced code-reuse attacks. Unfortunately, enforcing it without access to source code is hard in practice. The reason is that it requires an accurate Control-Flow Graph (CFG) and extracting such CFG from binary code is an undecidable problem. As a result, most existing binary-level CFI implementations base their invariants on an *approximation* of the CFG which leaves enough wiggle room for an attacker to launch successful exploits [10], [11], [15], [18], [19], [27].

While it may be possible to stop some advanced attacks using a perfect shadow stack implementation [9], there is one class of attacks for which there is no existing defense at the binary level whatsoever. This class of *function-reuse* attacks, exemplified by Counterfeit Object-Oriented Programming (COOP) [26], chains together calls to *existing* functions through *legitimate* callsites. This strategy preserves the integrity of the shadow stack, while abusing the overapproximation of the extracted CFG to call the *wrong* functions from these callsites. This attack is powerful since it allows for exploits that integrate smoothly with legitimate code execution. Unless there is deep knowledge of the C++ class hierarchy semantics, which we can only extract if we have the source code [13], it is hard to tell a COOP exploit apart from a legitimate code sequence [26]. Lacking a handle on the functions that a callsite may target leaves all the existing binary-level CFI solutions unable to stop COOP attacks.

In this paper, we revisit binary-level protection in the face of COOP attacks and follow-up improvements [13]. We explore to what extent we can narrow down the set of possible targets for indirect callsites and stop exploitation at the binary level. Our conclusion is *not* that all possible attacks can be stopped: even the tightest CFI solutions *with* access to source code are unable to guarantee perfect protection against all possible attacks [9]. Nevertheless, we demonstrate that *TypeArmor*, our binary-level protection prototype, can stop all COOP attacks published to date and significantly raise the bar for an adversary. Moreover, *TypeArmor* provides strong mitigation for many types of code-reuse attacks (CRAs) for programs binaries, without requiring access to source code. As researchers have shown that it is easy to bypass existing binary-level CFI defenses [10], [11], [15], [18], [19], [27], this is a significant improvement.

TypeArmor incorporates a forward-edge CFI strategy that relies on conservatively reconstructing both callee prototypes and callsite signatures and then uses this information to enforce that each callsite *strictly* targets matching functions only. For example, *TypeArmor* disallows an indirect call that prepares fewer arguments than the target callee consumes. Additionally, *TypeArmor* incorporates a novel protection policy, namely CFC (*Control-Flow Containment*), which further reduces the possible target set of callees for each callsite. CFC is based on the observation that, if binary programs adhere to standard calling conventions for indirect calls, undefined arguments at the callsite are not used by *any* callee by design. *TypeArmor* trashes these so-called *spurious arguments* and thus breaks all published COOP and improved COOP-like exploits. These exploits all chain virtual method calls that disrespect calling conventions to achieve convenient data flows between gadgets [13]. CFC eliminates these data flows via unused argument registers and thus stops such exploitation attempts.

Current binary-level solutions enforce "loose" forward-edge CFI policies, often allowing control transfers from any valid callsite to any valid referenced entry point [33], [34]. In the best case, existing policies only reduce the target set by removing all entry points of other modules unless they were explicitly exported or observed at runtime [24]. In contrast, *TypeArmor* matches up indirect callsites with a more precise target set in a many-to-many relationship. It relies on use-def analysis at all possible callees to approximate the function prototypes, and liveness analysis at indirect callsites to approximate callsite signatures. This effectively leads to a more precise CFG of the binary program in question, which could also be used by existing mitigation systems to amplify their (context-insensitive) invariants (e.g., PathArmor [30]).

Can TypeArmor defend against any exploit? No. *TypeArmor* protects only forward edges at the binary level. As shown by previous work, a backward-edge protection mechanism (e.g., a shadow stack [14] or context-sensitive CFI [30]) is still essential to ensure the integrity of return addresses at runtime [9], [18]. In this paper, we assume an ideal backward-edge protection mechanism such as a shadow stack with no design faults [12]. *TypeArmor* complements such backward-edge protection by countering attacks that take place without violating the integrity of the return path. Specifically, *TypeArmor* provides strong (but not infallible—given also the fundamental CFI limitations [9]) protection against COOP exploits as well as improved COOP-like exploits [13] and similar advanced attacks such as Control Jujutsu [16].

Is TypeArmor superior to approaches like IFCC/VTV and CPI? No. IFCC/VTV [29] and CPI [23] are strong source-level defenses which produce binaries that can resist control-flow hijacking attacks. Source-based techniques are more precise in using fine-grained program constructs (such as the C++ class hierarchy or generic data types) for mitigation purposes. However, there are still important reasons to study and improve binary-level defenses. First, the source code for many off-the-shelf programs is not always available. Second, real-world programs rely on a plethora of shared libraries and recompiling all shared libraries is not always possible. This is true even for purely open-source projects. For example, in VTV [29], the authors evaluate their system on ChromeOS, which includes legacy libraries. The authors had to manually whitelist these libraries, a task which is not trivial (certain code has to be annotated) and does not scale. Third, even if the source code of the libraries is available, recompiling big projects with dynamic dependencies is, again, a demanding task. Even state-of-the art defenses that enforce CFI policies at the source level such as SAFEDISPATCH [21] do not support dynamic libraries. Note that this is not a minor issue: mixing CFI-protected with non-protected code is impossible. If CFI is applied in just a portion of the CFG, crashes due to legitimate execution are possible. In contrast, with a binary-level solution, we can offer strong protection even if the source code is not available or when recompilation is not feasible (or desirable).

In summary, we make the following contributions in this paper:

- We introduce techniques to recover callsite signatures and callee prototypes for security enforcement purposes. Our techniques yield binary-level control-flow invariants which approximate the type-based invariants used in source-level solutions [29] and are thus much more precise than those used in prior binary-level CFI solutions [24], [33], [34].
- We demonstrate that fully-precise, static forward-edge CFI is inherently hard to achieve in a conservative fashion, due to the unavoidable precision loss when deriving traditional CFI-style target-oriented invariants at the binary level. To compensate for the precision loss, we complement our CFI strategy with a new technique termed *Control-Flow Containment* (CFC). CFC relies only on our callsite analysis to effectively *contain* code-reuse attacks. This approach improves the quality of control-flow invariants of traditional target-based approaches, overall resulting in a strict binary-level CFI strategy.
- We implement and evaluate *TypeArmor*, a new strict CFI solution for x86_64 binaries. Our experimental results demonstrate that *TypeArmor* can enforce much stronger forward-edge invariants than all the existing binary-level CFI solutions, while, at the same time, introducing realistic runtime performance overhead ($<$ 3% on SPEC).
- We show that our strict binary-level CFI strategy can mitigate advanced attacks in complete absence of source information or C++ semantics. For example, *TypeArmor* can stop all published COOP [26] exploits and their improvements [13].

The remainder of this paper is organized as follows. We start with a more detailed discussion of our main goal:

mitigating COOP-like attacks at the binary level. Section II provides a short introduction of how COOP works and Section III presents an overview of how *TypeArmor* is designed to mitigate COOP attacks. Section IV and V present *TypeArmor* internals. Section VI, Section VII, and Section VIII evaluate *TypeArmor*'s performance and security. Finally, Section IX surveys related work and Section X concludes the paper.

II. Motivation: Key Requirements for COOP

Counterfeit Object-Oriented Programming (COOP), proposed by Schuster et al. [26], is a novel attack technique that belongs to the class of code-reuse attacks (CRAs). While the core ideas have general applicability, the attack strategy described in [26] relies on Object Oriented Programming (OOP) principles and mainly targets C++ applications. In contrast to many proposed CRAs, COOP makes the exploit's control flow more akin to a benign execution flow. In this section, we summarize the technique with a focus on its key requirements: the ability to target unrelated virtual functions from an indirect callsite, and especially to pass data from one COOP gadget to another. In the next section, we show how *TypeArmor* impacts the attacker's possibilities to satisfy these requirements.

By exploiting a memory corruption bug, COOP diverts execution flow to a chain of *existing* virtual function calls (so called *vfgadgets*) via an *initial vfgadget*. In practice, an attacker can control said virtual function calls by injecting multiple, attacker-controlled *counterfeit objects* that reuse *existing* vtables in the binary. By choosing the correct object layout and *overlapping* multiple objects, an attacker can ensure intended data flows between different gadgets.

The original COOP paper [26], along with its improvement [13], proposes two main types of initial vfgadgets: (i) the main-loop gadget (ML-G) and (ii) the recursive gadget (REC-G). Such gadgets are responsible for dispatching the vfgadget chain using virtual function calls. The former depicts "[a] virtual function that iterates over a container [...] of pointers to C++ objects and invokes a virtual function of these objects" [26]. The latter, in turn, requires at least two consecutive virtual function calls on distinct (counterfeit) objects. The first call dispatches a vfgadget, whereas the last recurses into (any) REC-G.

Proper use of object overlapping may enable an attacker to pass data through object fields, if applicable. For example, one vfgadget may write to and another gadget then reads from, the same object field. In this paper, we refer to this strategy to pass data between vfgadgets as an *explicit* data flow. Schuster et al. found that cases that allow for explicit data flows are "rare in practice." [26]. Other approaches focus on the calling convention assumed by the indirect call that dispatches the vfgadgets. The ability to pass data to a vfgadget then depends on the choice of the ML-G, or REC-G, respectively. In the case of x86_64 calling conventions, the first six arguments are passed through registers (assuming System V ABI). These registers are *scratch* registers that are not preserved by a function. Consequently, if the ML-G or REC-G does not destructively update one of these registers in between virtual function calls, changes made to such a register by a vfgadget are implicitly passed to the next gadget. In other words, they represent an *implicit* data flow. Similar approaches for other platforms exist as well, for which we refer the reader to the original paper [26].

III. Overview

In this section, we first outline the threat model and assumptions under which *TypeArmor* operates. We then give a high-level overview of *TypeArmor* and discuss the impact of *TypeArmor*'s measures on COOP exploits.

A. Threat model and assumptions

We assume a common threat model where an attacker can read/write the data section and read/execute the code section of a vulnerable program. The program does not contain self-modifying code, $W \otimes X$ is in place, and the attacker is able to hijack the program's control flow by means of a memory-corruption vulnerability. We seek to defend against attacks with a binary-level version of (forward-edge) Control-Flow Integrity (CFI) [7]. In other words, our solution should support legacy binaries without access to source or debug symbols. In doing so, we focus on 64-bit binaries and analyze only function parameters that are passed via registers (those passed on the stack are conservatively handled). Depending on the ABI, this gives *TypeArmor* the capability to track at most 4 (in the case of Microsoft's x64 calling convention) or 6 (System V ABI) arguments. For simplicity, our implementation currently does not take floating-point arguments passed via `xmm` registers into consideration; future work may improve *TypeArmor* by extending static analysis to also include these registers. Nevertheless, as we show in Section VI, this still gives us enough information to stop even state-of-the art code-reuse attacks.

Obfuscated or hand-crafted binaries are out of scope and we assume an originating compiler that generally adheres to one of the standard calling conventions (to allow our static analysis to derive meaningful invariants), but can also occasionally resort to custom calling conventions for functions which are not externally visible due to standard compiler optimizations (which our analysis can conservatively handle). We discuss compiler optimizations in more detail in Section IV-B3, illustrating how *TypeArmor* can support optimizations from standard compilers and how it can be also extended to support optimizations from non-standard (or future) compilers. We stress that the current *TypeArmor* prototype works on stripped binaries that have been compiled using different optimization levels (namely `-O0`, `-O1`, `-O2`, and `-O3`).

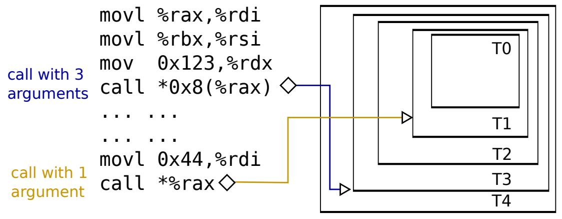
B. TypeArmor: Invariants for Targets and Callsites

TypeArmor deploys a combination of two type-based control-flow invariants, resulting in a strict forward-edge protection strategy: *target-oriented invariants* and *callsite-oriented invariants*. Target-oriented invariants are based on traditional CFI policies [7], but callsite-oriented invariants have not been explored for binaries before. Specifically, *TypeArmor* enforces callsite-oriented invariants through a novel containment technique which we term Control-Flow Containment (CFC). As noted above, extracting complete function and callsite type information at the binary level is hard in practice, and impossible in the general case. Therefore, *TypeArmor* relies on a *relaxed* form of type information (argument count and return value use), and enforces a many-to-many type-based matching strategy between callsites and targets. *TypeArmor* applies such type-based invariants, inspired by source-level CFI techniques [29], at the binary level for the first time.

In particular, *TypeArmor* ensures that indirect callsites that set at most max arguments cannot target functions that use more than max arguments. For instance, if *TypeArmor* finds a callsite that prepares at most 2 arguments, it ensures that the callsite can never jump to a function that consumes 3 targets or more. Additionally, *TypeArmor* ensures that indirect callsites that expect a return value (non-void callsites) can never jump to a callee that does not prepare such value (void functions). Enforcing such invariants at the binary level is challenging and subject to the precision of argument count and return use information derived by static analysis at *both* the callsite and at the target function.

While CFI's target-oriented invariants seek to identify the target set for each callsite, CFC follows a completely target-agnostic approach and thus is subject to the precision of argument count information *only* at the callsite. CFC relies on callsite-oriented invariants to scramble all the unused function arguments at every callsite, so that illegal (type-unsafe) function targets are not inadvertently exposed to stale (and potentially attacker-controlled) arguments. Similarly, at the callee, CFC is caller-agnostic and relies on liveness analysis to detect void functions. For these, *TypeArmor* scrambles unused return registers before the function returns. This strategy disrupts many type-unsafe function argument reuse attempts, which are required by existing COOP exploits. We include a formal definition of the invariants used by *TypeArmor*'s CFI and CFC in Appendix A.

Note that, in order to be conservative and support existing program functionality, *TypeArmor*'s callsite analysis is may only report an *overestimation* of the number of prepared arguments, while the callee analysis should report only *underestimation*. As an example, consider a callsite cs that prepares 3 arguments and a callee f that consumes 3 arguments. *TypeArmor* may detect that cs prepares 4 arguments and f only uses 2 arguments. *TypeArmor*'s invariants dictate



```
movl %rax,%rdi
movl %rbx,%rsi
mov  0x123,%rdx
call *0x8(%rax)
... ...
... ...
movl 0x44,%rdi
call *%rax
```

Figure 1. Hierarchical structure of binary-level function types. T_i is a set of functions that take i or less arguments. The first indirect call instruction (with 3 prepared arguments) may call functions in T_3 (which also include functions that are in T_2, T_1, and T_0), while the second indirect call instruction (preparing only 1 argument) may only target functions that are in T_1 (which includes those that are in T_0).

that, in this scenario, cs is still allowed to call f. Examples of how callsite overestimation and callee underestimation occur are further discussed in Section IV.

TypeArmor uses static analysis results to enforce control-flow invariants at runtime. The enforcement component relies on binary rewriting supported by the Dyninst binary analysis framework [8] to enforce CFI, CFC, or both (default configuration).

1) CFI: TypeArmor relies on the caller-to-callee mapping derived by our target-oriented invariants analysis. For this purpose, *TypeArmor* instruments each function according to its type and each indirect callsite to check if it calls the appropriately typed function. In contrast to source-level type-based CFI solutions [29], which benefit from one-to-one (i.e., precise function signature) mappings to detect type-incompatible targets, *TypeArmor* relies on a many-to-many mapping to sidestep the problem of identifying precise function signatures at the binary level—infeasible in general [25]. This strategy effectively results in a hierarchical function type structure when checking target-oriented invariants, as exemplified in Figure 1. As shown in the figure, the first callsite (at the top) passes 3 arguments to the callee, which thus belongs to set T_3 (also including sets T_0, T_1 and T_2). The second callsite (at the bottom), in contrast, passes only 1 argument to the callee and thus requires that the callee belongs to the set T_1 (also including T_0).

Note that the invariant that a non-void callsite cannot call a void function (omitted from Figure 1 for simplicity) doubles the number of function types: the set of functions that a particular non-void callsite may target is a subset of the possible targets for void callsites. This is because, at the binary level, it is only possible to determine potential non-void callsites. If our analysis finds that a callsite is not non-void, it cannot guarantee that this is a void callsite (the caller may call a non-void function, but never use its return value).

2) CFC: TypeArmor relies on the caller-to-type mapping derived by the callsite-oriented invariants analysis. For this purpose, it instruments each indirect callsite to scramble unused arguments before transferring control to the callee and instruments each void function to scramble unused return arguments before transferring control back to the caller.

A thorough analysis of *TypeArmor*'s static analysis is presented in Section IV, while we discuss the runtime component in Section V.

C. TypeArmor's Impact on COOP

TypeArmor's CFC enforces a maximum number of arguments prepared at a callsite and scrambles the unused registers. This severely impacts the ability of an attacker to enable data flow between gadgets.

As discussed in the original COOP paper [26], data flow via object fields is hard to achieve in practice due to a lack of useful gadgets. Instead, in the case of the x86_64 System V ABI, Schuster et al. suggest using unused argument registers to achieve data flow between indirect calls in the ML-G or REC-G, respectively. This only works if the invoking gadget does not update the register destructively. However, CFC is *explicitly* designed to introduce destructive updates of unused argument registers before an indirect call and mitigates this data-passing strategy. Furthermore, *TypeArmor*'s CFI implementation reduces the target set of the virtual function calls by the main-loop and recursive gadgets considerably. It prohibits any forward edges to functions that expect more arguments than the callsite prepares.

Needless to say, both aspects rely on the accuracy of *TypeArmor* in terms of callsite coverage in general and argument count identification for both callsites and target functions. Hence, implementing *TypeArmor* at the binary level is challenging from a research point of view and never as accurate as source-level solutions. However, we will show that it is effective in practice. In the next two sections, we look at the static analysis and dynamic enforcement of *TypeArmor*'s invariants.

IV. Static Analysis

Static analysis in *TypeArmor* seeks to detect (i) the maximum number of prepared arguments at indirect callsites, (ii) the minimum number of consumed arguments at possible callees, and (iii) the preparation (callees) and expectation (callsites) of return values. Since *TypeArmor* targets binaries, the analysis works on disassembled code. For this purpose, we leverage the Dyninst binary analysis framework which is capable of constructing Control-Flow Graphs (CFGs) for both program binaries and libraries [8].

A. Callee Analysis

We use static analysis to determine the argument count at the callee side. Given a set of address-taken (AT) functions[1], *TypeArmor* iterates over each function and performs a custom inter-procedural *liveness* analysis [22]. The analysis focuses on collecting state information on registers to determine if they are used for passing arguments or not. For a given path of instructions or basic blocks according the CFG, a register can be in one of the following states: *read-before-write* (R) (data are always read from this register before new data are written to it), *write-before-read* (W) (this register is always written to before it is read), or *clear/untouched* (C) (this register is never read or written to). The state of a particular basic block contains the combined register state for all argument registers. The analysis starts at the entry basic block of an AT function and iterates over the instructions to determine the usage of registers. If all argument registers are either R or W, the analysis terminates. However, if at least one register is in a C state, a recursive *forward analysis* starts until the block has no outgoing edges. Note that the analysis takes special care about variadic functions, which we discuss in Section IV-A4.

1) Forward Analysis: A recursive analysis loops over all outgoing edges of the basic block to get a pointer to the next basic block to analyze. We distinguish between direct calls, indirect calls, return instructions, and regular outgoing edges (e.g., jump instructions). Depending on the edge type, different operations are performed.

Direct calls: For direct calls, the next basic block to analyze is the entry block of the target function. We also retrieve the *fall-through* basic block for this instruction, which is the block to be executed after the direct call returns. For each direct call, we push the fall-through block on a stack that *TypeArmor* maintains, which we later use to analyze return instructions (see below). In the case of direct calls that never return (e.g., calls to functions that exit), we do not retrieve a fall-through block. We detect such calls by checking whether they target a known function that exits (e.g., `exit@plt`). This analysis is again recursive so that we can correctly wrappers around `exit` as non-returning functions.

Indirect calls: The analysis cannot statically infer the target of the indirect calls and we thus have to be conservative. We assume that the target writes all arguments and stop the recursion, transforming all remaining *clear* registers into a *write-before-read* state.

Returns: For return instructions, we pop a fall-through basic block from the stack and use it as the next basic block in the analysis. An empty stack indicates the end of the analyzed function and terminates the recursive analysis.

Other: We handle other edge types (including indirect jumps, for which we rely on Dyninst to resolve its targets) in the same way: the targets of the edge are set as the next basic blocks in the analysis.

Finally, to avoid loops during the analysis, we keep track of all blocks analyzed so far. When the analysis is about to recurse, we check whether we already analyzed the next basic block, and if so, continue with the next edge. In

[1]A function f is defined to have its address taken if there are one or multiple instructions in the binary that load the entry point of f into memory. By definition, indirect calls can only target AT functions.

addition, we use a cache to avoid multiple analysis passes on the same basic block. Notice that the latter is just an optimization for speeding up the analysis (which is offline), and it does not affect the accuracy of the results.

2) Merging Paths: The value returned by *TypeArmor*'s recursive forward static analysis for a basic block B, which has n outgoing edges, provides us with a set of states S_i $(i = 1, 2, \ldots n)$. These states represent argument usage information for each path following edge i. Each state is represented by a vector composed by the state of each one of the six argument registers. *TypeArmor* combines these states into a *superstate* S that denotes the argument liveness for any path following B. For this purpose, we use a conservative policy that mandates that the state for argument register c in S can only be R if the state for c is R for all states S_i $(i = 1, 2, \ldots n)$ following B. In other words, states W and C always supersede R, but both (W and C) are neutral with each other. After computing S, *TypeArmor* combines it with S_B, the state information for B. The merging policy here is slightly different in that states other than C in S_B always supersede states in S. This is because B is executed *before* any of its following basic blocks. For an actual example of how path merging works, please refer to Figure 3 on page 7 and the explanation in Section IV-A6.

3) Argument Count: Once the recursive analysis converges to a definite state for the entry basic block of a function, the argument count is set using the highest argument register that is marked as R. For instance, the System V ABI uses **rdi**, **rsi**, **rdx**, **rcx**, **r8**, and **r9**, as arguments registers. Therefore, if **r9** has a read-before-write state, we conclude that this particular function expects *at least* 6 arguments. If **r9** is W or C, then **r8** is examined, and so on.

4) Variadic Functions: Since variadic functions can take any number of arguments and thus may use all argument registers, variadic arguments may end up being passed in both CPU registers and memory (via the stack). To support easy manipulation of variadic arguments, modern compilers tend to move all the variadic arguments onto the stack in successive order upon entry of a variadic function. To make sure that the forward static analysis does not erroneously interpret the moving of argument registers to the stack as read-before-write operations (and conclude that this function expects more arguments than are defined), *TypeArmor* identifies variadic functions by means of pattern matching.

A function is labeled to contain n possible variadic arguments iff (i) a series of n argument registers, starting from the last argument register (**r9** for the System V ABI), are marked R, (ii) these reads occur in the same basic block (and in the appropriate order), and (iii) the arguments are written on the stack. If *TypeArmor* finds that a function contains n argument registers, it limits the maximum number of arguments for this function as computed by our forward analysis to $max - n$, where max is defined to be the maximum numbers of arguments that can be passed via registers (6 for the System V ABI). Figure 2 illustrates the operation of *TypeArmor*'s variadic function detection mechanism using as an example the **ngx_snprintf** function.

ngx_snprintf

```
u_char * ngx_cdecl
ngx_snprintf(u_char *buf, size_t max, const char *fmt, ...)
{
    u_char   *p;
    va_list   args;

    va_start(args, fmt);
    p = ngx_vslprintf(buf, buf + max, fmt, args);
    va_end(args);
    return p;
}
```

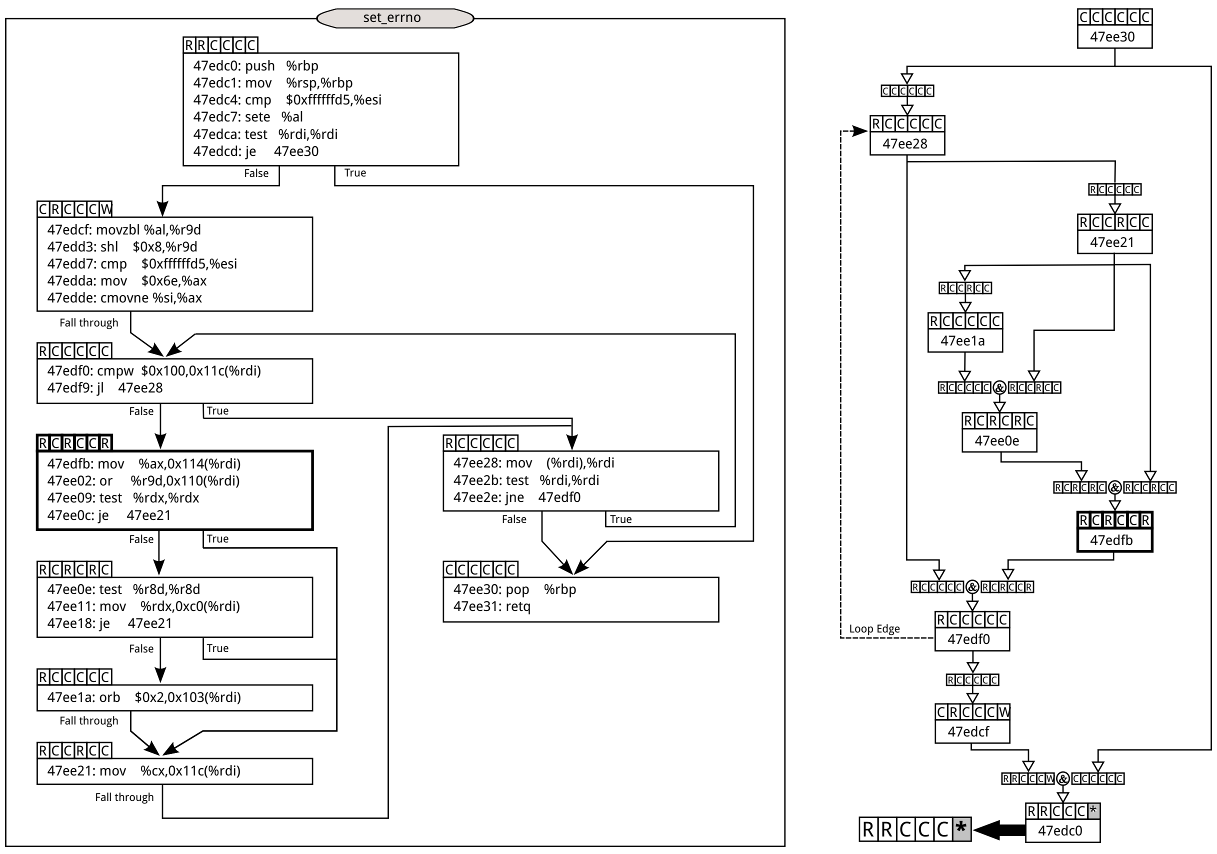
ngx_snprintf

```
40f310: push   %rbp                      40f34c: mov   %r8,-0xb0(%rbp)
40f311: mov    %rsp,%rbp                 40f353: mov   %rcx,-0xb8(%rbp)
40f314: sub    $0xd0,%rsp                40f35a: lea   -0xd0(%rbp),%rax
40f31b: test   %al,%al                   40f361: mov   %rax,-0x10(%rbp)
40f31d: je     40f345                    40f365: lea   0x10(%rbp),%rax
40f31f: movaps %xmm0,-0xa0(%rbp)         40f369: mov   %rax,-0x18(%rbp)
40f326: movaps %xmm1,-0x90(%rbp)         40f36d: movl  $0x30,-0x1c(%rbp)
40f32d: movaps %xmm2,-0x80(%rbp)         40f374: movl  $0x18,-0x20(%rbp)
40f331: movaps %xmm3,-0x70(%rbp)         40f37b: add   %rdi,%rsi
40f335: movaps %xmm4,-0x60(%rbp)         40f37e: lea   -0x20(%rbp),%rcx
40f339: movaps %xmm5,-0x50(%rbp)         40f382: callq 40eb90
40f33d: movaps %xmm6,-0x40(%rbp)         40f387: add   $0xd0,%rsp
40f341: movaps %xmm7,-0x30(%rbp)         40f38e: pop   %rbp
40f345: mov   %r9,-0xa8(%rbp)            40f38f: retq
```

Figure 2. Variadic function detection searches for a basic block that performs read-before-write operations on a series of argument registers in consecutive order (either from lowest to highest in the case of **gcc**, or vice versa for **clang**) without other instructions in between. Observe in this particular example a group of instructions near the address **0x40f345**. The last three argument registers, namely **r9**, **r8**, and **rcx**, are moved (read-before-write) to the stack through instructions contained in a single basic block and in a specific order. Thus, this is a variadic function that uses its last three argument registers to hold variadic arguments.

We tested our variadic function detection mechanism against binaries compiled with both clang and gcc and found zero cases where a variadic function was mistakenly detected as a regular one. We did, however, observe a handful of cases where a function was wrongly detected as accepting a variadic number of arguments, leading to an underestimation of the number of arguments used (see also Section VIII).

5) Conservativeness: A key property of the analysis performed by *TypeArmor* at the callee is that it is conservative and therefore underestimation of the argument count is possible. Some interesting cases are: (i) instructions that perform a read and write on the same register (e.g., **xor %rdi,%rdi** or **neg %r9**), (ii) underestimated callees deriving from functions mistakenly detected as variadic, (iii) functions with *many* arguments (some of them passed through the stack), (iv) analyzed paths that contain further indirect calls, and (v) callbacks that do not actually use *all* arguments. We stress that *TypeArmor* correctly handles case (i) and assigns the register either the state R or W depending on the used instruction (e.g., **xor %rdi,%rdi** is W and **neg %r9** is R). For (v), *TypeArmor* yields better results than a source-level analysis. As an example, consider a generic signal handler implementation, where the signal number is



```
47edc0: push  %rbp
47edc1: mov   %rsp,%rbp
47edc4: cmp   $0xffffffd5,%esi
47edc7: sete  %al
47edca: test  %rdi,%rdi
47edcd: je    47ee30
```

```
47edcf: movzbl %al,%r9d
47edd3: shl   $0x8,%r9d
47edd7: cmp   $0xffffffd5,%esi
47edda: mov   $0x6e,%ax
47edde: cmovne %si,%ax
```

```
47edf0: cmpw  $0x100,0x11c(%rdi)
47edf9: jl    47ee28
```

```
47edfb: mov   %ax,0x114(%rdi)
47ee02: or    %r9d,0x110(%rdi)
47ee09: test  %rdx,%rdx
47ee0c: je    47ee21
```

```
47ee0e: test  %r8d,%r8d
47ee11: mov   %rdx,0xc0(%rdi)
47ee18: je    47ee21
```

```
47ee1a: orb   $0x2,0x103(%rdi)
```

```
47ee21: mov   %cx,0x11c(%rdi)
```

```
47ee28: mov   (%rdi),%rdi
47ee2b: test  %rdi,%rdi
47ee2e: jne   47edf0
```

```
47ee30: pop   %rbp
47ee31: retq
```

(a) CFG

(b) Register merging

Figure 3. Callee analysis for `void set_errno(address_item *addrlist, int errno_value, uschar *msg, int rc, BOOL pass_message)` (of Exim). Observe how merging paths works. The basic block starting at `0x47edfb` (emphasized in bold) has state $S_B = (R, C, R, C, C, R)$, since `rdi`, `rdx`, and `r9` are read. There are two incoming states to this block, namely $S_1 = (R, C, R, C, R, C)$ and $S_2 = (R, C, C, R, C, C)$, which are combined to a superstate $S = (R, C, C, C, C, C)$ (note that C always supersedes). Finally, the superstate is combined with the block state, but this time R supersedes and hence the output state is $(R, C, C, C, C, C) \wedge (R, C, R, C, C, R) = (R, C, R, C, C, R)$. The final state of all analyzed blocks is $(R, R, C, C, C, *)$, where the $*$ denotes that C does not supersedes W or vice versa.

always passed—and, thus, at least one argument is expected to be passed to the callee—but not necessarily used by the handler, something *TypeArmor* can accurately infer.

6) Example of Operation: To illustrate how the analysis at the callee works, consider the `set_errno` function (taken from Exim) in Figure 3. The entry basic block contains read operations on the first two argument registers (`rdi` and `esi`). At this point, the analysis cannot infer other possible arguments, but it can certainly proceed further. Based on the outcome of the conditional operation at address `0x47edcd` there are two available paths. In case the conditional check is *False*, the fall-through basic block at offset `0x47edcf` should be followed, otherwise, control should be transferred to address `0x47ee30`. The latter path simply returns and thus ends the function without any additional read-before-write operations. Since the analysis is conservative, this short path is sufficient to conclude that a *minimum* of two arguments are used by this function.

To illustrate *TypeArmor*'s forward merging process, we include a complete merge graph for the `set_errno` function in Figure 3(b). Since merging is a backward process, the figure shows the CFG "up-side-down". As an example merging step, consider the basic block starting at `0x47edfb` which has $S_B = (R, C, R, C, C, R)$ (`rdi`, `rdx`, and `r9` are read). There are two incoming states to this block, namely $S_1 = (R, C, R, C, R, C)$ and $S_2 = (R, C, C, R, C, C)$, which are combined to a superstate $S = (R, C, C, C, C, C)$ (notice that C always supersedes). Finally, the superstate is combined with the block state, but this time R supersedes, and hence the output state is $(R, C, C, C, C, C) \wedge (R, C, R, C, C, R) = (R, C, R, C, C, R)$.

B. Callsite Analysis

TypeArmor iterates over each indirect callsite and performs a backward static analysis—a variant of classical *reaching definition* analysis [22]—to detect the prepared argument count at a particular callsite. The backward static analysis collects state information on all possible argument registers, but unlike our forward static analysis (Section IV-A), it only accounts for registers that are either set (S) or not (T, *trashed*). In particular, *TypeArmor* starts the analysis at the basic block that contains the indirect call, and iterates over preceding instructions for determining whether argument registers are S or T. If all argument registers are S, *TypeArmor* stops the analysis and assumes that the callsite uses the maximum number of arguments. If some arguments cannot be considered either S or T and the basic block has incoming edges, *TypeArmor* starts a recursive backward analysis.

1) Backward Analysis: Direct calls, returns, and other incoming edges are distinguished in the same fashion as in the callee analysis (see Section IV-A). For direct calls, the preceding basic block to analyze next is the basic block where the direct call originated. This means that if the backward analysis reached the entry block of the function containing the inspected callsite, an inter-procedural backward analysis at all the callers of this function is initiated. Return edges during backward analysis indicate that the currently analyzed basic block has a predecessor that performs a function call. Thus, at this point, traversing further in this path is stopped and all remaining argument registers are marked as T: we assume that argument registers are always reset between two calls. This means that analysis is terminated and the state of this basic block is returned. Note that since indirect call targets cannot be resolved statically, there are no indirect call edges.

2) Merging Paths: Path merging for the backward static analysis is relatively straightforward: for all collected states of the incoming basic blocks, T always supersedes S (arguments must be set on *all* paths). Similar to the forward analysis, once the recursive analysis is finished, the number of prepared arguments is set based on the states of the last write operations.

As an example, consider an indirect callsite cs that is reachable by two basic blocks b_1 and b_2, both of which are preceded by another indirect call instruction. If the backward analysis finds that b_1 writes to (sets) arguments register **rdi**, **rsi**, and **rdx** (the first three argument registers), while b_2 only sets **rdi**, *TypeArmor* concludes that cs prepares *at most* one argument.

3) Compiler Optimizations: TypeArmor's current implementation of backward static analysis may yield false conclusions (underestimation of number of prepared arguments) if the compiler deploys (inter-procedural) redundant argument register write elimination. Two examples of such optimization, which the compiler may perform at code generation time, are shown in Figure 4. Figure 4(a) shows how an inter-procedural write elimination pass may omit the second **mov $0x1,%rdi** instruction (depicted in red), since **rdi** has already been set to the same constant value in **foo**. Figure 4(b) shows a similar optimization instance, however eliminating writes across functions.

```
foo(void)
    mov $0x1, %rdi

bar(int arg1)
    ...

main(void)
    foo()
    mov $0x1,%rdi
    call bar
```

(a)

```
bar(int arg1)
    ...

foo(int arg1)
    ...

main(void)
    mov $0x1,%rdi
    call foo
    mov $0x1,%rdi
    call bar
```

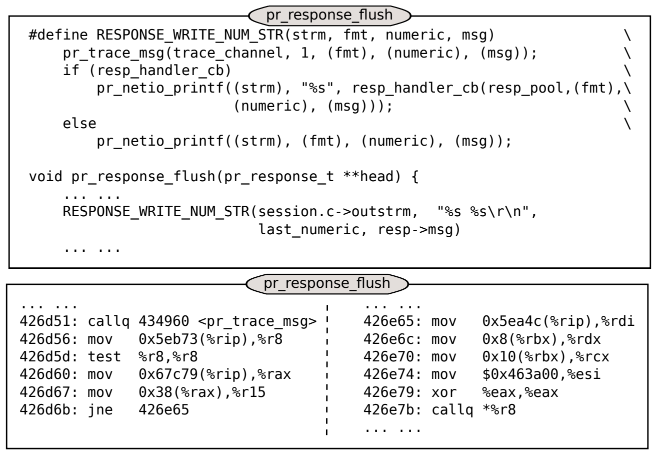
(b)

Figure 4. Two variants of (inter-procedural) redundant argument register write elimination. In both code snippets, a compiler optimization pass may omit the second **mov $0x1,%rdi** instruction (depicted in red). In (a), **%rdi** has been set to 1 in **foo** already and is never modified before the call to **bar**, making the second **mov** operation redundant. A similar scenario occurs in (b): the compiler may conclude that **rdi** has been set before the call to **foo**, and was never modified before the second **mov** instruction.

After analyzing the source code of two popular compilers (clang and gcc), we found no evidence of the presence of above optimization. Moreover, as we show in Section VIII, a thorough comparison of clang-generated binaries against LLVM ground truth, across different optimization levels, did not reveal any underestimation of prepared arguments. These results confirm that the assumption that standard compilers always explicitly (re)set argument registers after a direct (and not only indirect) call is safe. For nonstandard or future compilers that may deploy inter-procedural write elimination optimizations, a possible solution is to continue backward analysis from indirect callsite cs_2 until another indirect call cs_1 is found instead of a direct call: since the compiler does not know the target of cs_1 (or else it would have been a direct call), it shall reset all required arguments for cs_2, making our backward analysis between indirect calls safe by design.

4) Conservativeness: As with the callee analysis, *TypeArmor*'s callsite analysis should be conservative and therefore only allow for overestimation of the argument count. An interesting case to consider is how the analysis performs for indirect callsites inside *wrapper* functions. Such functions may not need to reset all argument registers, but simply 'pass them through' directly from its caller. However, if the wrapper has its address taken, and is only called through indirect functions, our backward analysis fails to find any incoming edges to the basic blocks and must give up. In order to be conservative, *TypeArmor* then decides that the callsite inside the wrapper prepares the maximum number of arguments.

To improve static analysis results for callsites, we complement *TypeArmor* to accept profiling data to improve its CFG. Consider above scenario of an indirect callsite inside a



```
#define RESPONSE_WRITE_NUM_STR(strm, fmt, numeric, msg)             \
    pr_trace_msg(trace_channel, 1, (fmt), (numeric), (msg));         \
    if (resp_handler_cb)                                             \
        pr_netio_printf((strm), "%s", resp_handler_cb(resp_pool,(fmt),\
                        (numeric), (msg)));                          \
    else                                                             \
        pr_netio_printf((strm), (fmt), (numeric), (msg));

void pr_response_flush(pr_response_t **head) {
    ... ...
    RESPONSE_WRITE_NUM_STR(session.c->outstrm,  "%s %s\r\n",
                           last_numeric, resp->msg)
    ... ...
```

```
... ...                                     ... ...
426d51: callq 434960 <pr_trace_msg>      426e65: mov   0x5ea4c(%rip),%rdi
426d56: mov   0x5eb73(%rip),%r8          426e6c: mov   0x8(%rbx),%rdx
426d5d: test  %r8,%r8                    426e70: mov   0x10(%rbx),%rcx
426d60: mov   0x67c79(%rip),%rax         426e74: mov   $0x463a00,%esi
426d67: mov   0x38(%rax),%r15            426e79: xor   %eax,%eax
426d6b: jne   426e65                     426e7b: callq *%r8
                                            ... ...
```

Figure 5. Partial disassembly of the **`pr_response_flush`** function in ProFTPD that illustrates the working of our callsite analysis. The indirect call instruction at offset `0x426e7b` maps to the call to **`resp_handler_cb`**. *TypeArmor*'s backward analysis finds that the basic block that ends with this indirect call writes to the first four arguments and thus continues analysis at incoming basic blocks. Only one such block exists and it performs a write operation on the fifth argument register (**`test %r8,%r8`**). Since the path that leads to this block ends with a call to **`pr_trace_msg`**, *TypeArmor* concludes that the indirect call **`callq *%r8`** prepares *at most* 5 arguments.

wrapper function. If a profile run finds an edge from another callsite to the wrapper, our static analysis can continue its backward analysis and possibly reduce the number of allowed arguments.

5) Example of Operation: Consider **`pr_response_flush`** from ProFTPD, which is depicted in Figure 5. Notice the indirect call located at offset `0x426e7b` which maps to **`resp_handler_cb`**; a variadic function that takes two fixed arguments. By analyzing the basic block, we infer that at least four argument registers are *live* (due to the 4 **`mov`** instructions). Since there is no information for the two additional argument registers (`r8` and `r9`), through recursive analysis, *TypeArmor* discovers all basic blocks directly pointing to `0x426e65`. For this particular scenario, one such block exists, starting at `0x426d56`. This block contains an instruction that moves a value into register `r8`, therefore this callsite is marked to hold a fifth argument. For inferring if `r9` is used as well, the analysis further proceeds and finds one basic block pointing to `0x426d56`. This block contains a return edge from **`pr_trace_msg`**, thus `r9` cannot be used as an argument register. As a result of the backward analysis, *TypeArmor* concludes that the callsite `0x426e7b` prepares *at most* five arguments, one more than the actual number of prepared arguments (**`strm`**, **`fmt`**, **`numeric`** and **`msg`**).

C. Return Values

Adding information about return value usage improves the precision of *TypeArmor*'s CFI implementation: if we find a callsite that expects a return value (a non-void callsite), it should never target a callee that does not prepare a return value (void functions). Extracting return usage information from callsites and callees is similar to the previously described callee and callsite analysis and is again conservative: a void callsite is allowed to target both void and non-void callees.

1) Non-void Callsites: The detection of non-void callsites (i.e., callsites that expect a return value), is done by searching for *read-before-write* operations on the the register that holds return values (**`rax`** for the System V ABI). In essence, we apply the forward analysis as used by our callee analysis, but now starting from the callsite, and only for **`rax`**. The difference is that we keep the analysis intra-procedural in order to remain conservative.

2) Void Callees: We detect void functions by applying the previously described backward analysis at the *exit points* of a function (exit points are basic blocks that end with a **`ret`** instruction). The backward analysis only searches for write operations on **`rax`** which may indicate a set return value.

In order not to break programs, our non-void callsite analysis is conservative and marks a callsite as void (allowing it to target both void and non-void functions) if no read-before-write on return registers is found (the callsite may pass the return value to a caller directly). Similarly, conservativeness at the callee results in an underestimation of the number of void functions: the compiler may use return registers as scratch registers, which we cannot detect by looking at disassembled instructions only. We describe the precision of our return value analysis in Section VIII.

Note that if a particular ABI specifies that multiple registers may be used to hold return values (like the System V ABI allows callees to use the register pair **`rax:rdx`**), *TypeArmor* could be extended to perform a similar analysis on those as well.

V. Runtime Enforcement

In this section, we describe how *TypeArmor* uses the results from the static analysis, discussed in Section IV, to provide security guarantees at runtime. During application load time, *TypeArmor*'s runtime component instruments the application's binary and loaded libraries to enforce our CFI and CFC policies. We achieve this by adding integrity and containment code at the forward edges and labels at function entry points. The runtime component can be split in three parts: (i) shadow code memory preparation, (ii) CFI enforcement, and (iii) CFC enforcement.

A. Shadow Code Memory Preparation

At every library load, this part of *TypeArmor*'s runtime component allocates memory to store instrumented code, dubbed *shadow code* (as implemented by Dyninst [8]). The shadow code is essentially a copy of the original code that also contains the instrumentation of the callsites. Program execution is performed using the instrumented shadow code.

Whenever we reach an indirect callsite during normal program execution, the instrumentation code at this location performs an integrity check between the type of the callsite and the type of the callee. If the types are compatible with each other, the callsite branches to the callee. Note that the branch target of the callsite still remains in the original code region. Therefore, we replace the beginning of each AT function in the original code with a jump instruction that jumps to the corresponding function in the shadow code region.

We perform the integrity check by retrieving and processing the function's label, located right before the function entry point in the original code region. Using this strategy, we do not have to ensure that our label does not overwrite code since the code that is executed is located in the shadow code region. Choosing the right label is not an easy task because we have to verify that this label does not occur at locations other than AT functions.

We tackle this problem with an approach similar to the pointer masking technique discussed by Wahbe *et. al.* [31]. After moving all the code to the shadow code region, the unused locations in the original code region (i.e., all but AT function entry points) are filled with trap instructions[2]. Furthermore, during program execution, the integrity check that is performed at indirect callsites (further discussed in Section V-B1) first masks the target address, so it can only point to the original code region, before continuing the execution. Using this strategy, indirect callsites can only point to compatible AT functions.

Note that without the additional instrumentation for *type* compatibility checks, the implementation with the shadow code region results in a coarse-grained CFI solution for forward edges, in which indirect callsites can target **all** AT functions without any restrictions.

B. CFI Enforcement

TypeArmor instruments binaries for enforcing that callsites can only target functions with a compatible *type*. This essentially means (i) a callsite with a higher number of prepared arguments can target all the functions that any callsite with a lower number of prepared arguments can also target, but not vice versa, and (ii) a callsite that expects a return value can only target functions that return a value, whereas a callsite that does not expect a return value can target both functions that do and do not return a value. To implement these policies, *TypeArmor* has to instrument both, callsites and callees, based on the information collected through static analysis (see Section IV).

1) Callee instrumentation: We label each AT function (i.e., prepend it with a magic number) similar to the original instrumentation scheme of Abadi *et. al.* [7]. In the context of *TypeArmor*, there are seven possible labels (no arguments (0) to all arguments (6)), therefore, we use a 3-bit representation. In addition, we use one more bit to represent whether the function returns a value. We use 1 to encode *void* functions and 0 for functions that return a value. This is an important design decision for the callsite instrumentation, because callsites that expect a return value need to be handled in a special way (i.e., they can only target non-void functions) and this allows us to do it with just *one* extra instruction. This is further explained in Section V-B2.

In practice, we use a 4-byte label and encode the function type using four bits of the label. For the return type, we use the least significant bit and, for the number of arguments, the adjacent three bits of the label. For example, we represent the bits of a *void* function that has four arguments according to the static analysis as `1001`.

To have a unique combination of four bytes that does not occur at any other code location, we choose `0xCCCCCC40` as a base label and use the four least significant bits to encode the function type. This form is suitable because all unused bytes are set to the trap instruction with which also the original code region is filled (see Section V-A). The upper half of the least significant byte is set to four, because regardless of the value of the lower half of the byte, this byte assembles into the REX instruction prefix for the trap instruction[3]. Since REX has no effect when combined with the trap instruction, this label does not lead to valid targets for an attacker.

2) Callsite instrumentation: At each callsite, *TypeArmor*'s runtime component inserts a check to determine if the target is legal as per the CFI policy. It does so by retrieving the callee's label, decoding the type and checking if the result is compatible with the callsite. The instrumented check does the following:

1) Get the address of the target.
2) Mask the target address to force the callsite to point into the original code region.
3) Read the target's memory at target -4 to get the label.
4) Apply `xor` at the label with the value `0xCCCCCC40`. Note that we do not explicitly check if this part of the label was correct. If the label was incorrect, the check for the number of arguments (step 6) fails, since the result represents an unexpected value.
5) Only for callsites that expect a return value: make sure that the last bit is 0 (i.e., the target function does return a value) which is done by applying a right rotate by 1 bit on the label. Note that if the callsite targets a void function, the subsequent check fails, since the bit rotation results in a large value.
6) Using an unsigned comparison, check if the resulting value is below or equal to the (hardcoded) number

[2]Byte 0xCC is a trap instruction and disassembles to *int3*.

[3]In little endian, the label `0xCCCCCC40` would be represented as 0x40 0xCC 0xCC 0xCC in memory, which assembles to the code `REX INT3; INT3; INT3`.

of arguments the callsite is expecting. The range of possible values for callsites that expect a return value is $0 - 6$ and for callsites that do not expect a return value, the range is $0 - 13$. Note that the latter range also includes the return type bit.

As an example, consider the case where an indirect callsite which prepares four arguments and expects a return value tries to target a void-typed function that expects at least one argument. This function is assigned the label `0xCCCCCC43`. At the callsite (after masking the target address, retrieving the label, and xoring the label with `0xCCCCCC40`) a right rotate of 1 bit is performed, because the callsite expects a return value. This results in the value `0x80000001`. Subsequently, the check for the number of arguments fails, since the resulting value is larger than 4, i.e. the prepared number of arguments at the callsite.

C. CFC Enforcement

We enforce CFC by scrambling unused registers at indirect callsites. Using this strategy, we essentially enforce a zero percent underestimation rate at the callee, at the cost of losing the ability to detect ongoing attacks, but instead silently crashing. Similarly, CFC scrambles the unused return register **`rax`** at return instructions of void functions so that we eliminate overestimation of non-void callsites.

As an example, consider an AT function f that accepts five arguments, but for which *TypeArmor* conservatively concludes that it accepts at least two arguments. Now, consider an indirect callsite cs for which *TypeArmor* assumes that it sets no more than three arguments. Without enforcing CFC, cs is allowed to target f. By enabling CFC, *TypeArmor* instruments cs in such a way that the last three argument registers (i.e., **`rcx`**, **`r8`**, and **`r9`**) are initialized with a random values at the callsite. The used random values are generated and inserted into the instrumentation code during load time. Observe that this does not change the fact that cs is allowed to target f. What it does enforce, however, is that if f enters a path that uses the 4th and 5th argument registers, the program is likely to crash as their values are no longer valid. Notice that we use random values (precomputed at load time) to initialize the argument registers and not fixed ones (such as zero). This is on purpose to avoid the risk of attacks based on malicious control flows that leverage a known state of the argument registers.

VI. Mitigating Advanced Code-reuse Attacks

In this section, we discuss how effective *TypeArmor* is in stopping advanced code-reuse attacks (CRAs). Table I presents a short summary of recently published CRAs that rely on control-flow diversion and how *TypeArmor* addresses them. Note that *all* publicly available exploits that are not pure data-only attacks (like Control Flow Bending [9]) are successfully mitigated.

Table I
TypeArmor stops existing code-reuse exploits. Since *TypeArmor* specifically targets x86_64 binaries, the IE 32-bit COOP exploit is out of scope. Note that even without deploying CFC, *TypeArmor* stops all exploits.

Exploit	**Stopped?**	**Notes**
COOP ML-G [26]		
– IE (32-bit)	✗	Out of scope
– IE 1 (64-bit)	✓(CFI)	Argcount mismatch
– IE 2 (64-bit)	✓(CFI)	Argcount mismatch
– Firefox	✓(CFI)	Argcount mismatch
COOP ML-REC [13]		
– Chrome	✓(CFI)	Argcount mismatch, Void target where non-void was expected
Control Jujutsu [16]		
– Apache	✓(CFI)	Target function not AT
– Nginx	✓(CFI)	Void target where non-void was expected

In the following sections, we discuss advanced CRAs in more detail, COOP in particular. First, in Section VI-A, we analyze a set of server applications for COOP gadgets while *TypeArmor* is in place and explore if COOP is still possible. Next, in Section VI-B, we walk through practical COOP exploits for Internet Explorer, Firefox, and Chrome to show how *TypeArmor* stops these attacks. In Section VI-C, we discuss how *TypeArmor* stops Control Jujutsu exploits [16]. In Section VI-D, we discuss further possibilities of COOP exploitation. Finally, we conclude in Section VI-E with a discussion on pure data-only attacks such as those presented by Control-Flow Bending [9].

A. Effectiveness against COOP

Armed with the knowledge that COOP relies on unused argument registers to enable data flow between gadgets (Section II), we are interested in how many of those spurious arguments remain when *TypeArmor* is in place. We applied *TypeArmor*'s static analysis on a large set of server application binaries and compared results against ground truth obtained by LLVM (more details in Section VII). Table II shows, for each server application, (i) the number of indirect call instructions (*cs*), (ii) the number of callsites for which our analysis reports the *exact* number of prepared arguments as defined at the source level (*0*), and (iii) the number of callsites for which we overestimate the number of prepared arguments by 1, 2, $\ldots$ (*+N columns*).

From Table II, we conclude that *TypeArmor* is able to determine the exact number of prepared arguments for 103 out of 130 indirect callsites (geometric mean). While these numbers are fairly promising already, the missing 23 callsites are potentially dangerous and could still be used as the initial COOP gadget by an attacker. To investigate this further, we operated a heuristic search for all possible main-loop (ML-G) and recursive (REC-G) gadgets for each of the server applications. We depict overestimation results

Table II
ACCURACY OF *TypeArmor* COMPARED TO THE GROUND TRUTH FOR DIFFERENT SERVER APPLICATIONS. THE NUMBERS IN THE COLUMNS DEPICT HOW FAR OFF THE ANALYSIS IS IN TERMS OF NUMBER OF ARGUMENTS (I.E., HOW MANY ADDITIONAL ARGUMENTS ARE ERRONEOUSLY ASSUMED BY *TypeArmor*) FOR THE CALLSITES IN THE ANALYZED SERVER APPLICATIONS.

		Overestimation					
Server	*#cs*	*0*	*+1*	*+2*	*+3*	*+4*	*+5*
Exim	76	65	6	3	1	0	1
lighttpd	54	47	0	2	0	0	5
Memcached	48	41	3	2	0	2	0
Nginx	218	161	35	16	3	1	2
OpenSSH	134	130	4	0	0	0	0
ProFTPD	85	68	10	3	2	2	0
Pure-FTPd	10	8	1	0	1	0	0
vsftpd	4	2	2	0	0	0	0
PostgreSQL	491	392	52	22	9	3	13
MySQL	7532	5771	789	366	269	125	212
Node.js	2452	2113	226	37	25	10	41
geomean	130	103	15	11	6	5	10

for these possible gadgets in Table III (not completely unexpected, we only found reasonable gadgets in the C++ binaries—MySQL and Node.js).

For the main loop gadgets, *TypeArmor* accurately identified the argument count for 94% of the callsites in MySQL and for 95% in Node.js. Similarly, for the recursive gadgets, we identified the exact argument count in 86% (MySQL) and 96% (Node.js) of the cases. This means, however, that the remaining callsites may allow data to flow via the overestimated argument register(s) as identified by *TypeArmor*: these registers are not explicitly initialized with an argument value and may pass data set by one vfgadget to the next. As our automated gadget identification is not precise, we manually analyzed the remaining gadgets that might allow implicit data flow (data flow via spurious arguments). As noted in Section II, the registers in question might still be unusable for data flow due to destructive updates in between the indirect calls.

For the main loop gadgets in MySQL, we found five callsites that were mistakenly reported to have an overestimated argument (caused by the fact that LLVM IR blocks may still get optimized or shuffled when bitcode is lowered to machine instructions), leaving only five callsites with true overestimation. For Node.js, overestimation affects six callsites. Manually analyzing the reported gadgets, however, revealed that no implicit data flow is possible for these callsites. Results for recursive gadgets look equally promising: overestimation of prepared argument count occurred for 38 callsites in MySQL and two in Node.js. Manual analysis revealed that four gadgets were wrongly identified as REC-Gs, 13 could not set up an implicit data flow due to destructive updates, and for 23, CFC prevents data flow.

Table III
ACCURACY OF *TypeArmor* COMPARED TO THE GROUND TRUTH FOR DIFFERENT SERVER APPLICATIONS. THE VALUES ARE GIVEN IN RESPECT TO CALLSITES BELONGING TO A SPECIFIC TYPE OF GADGET.

		Overestimation					
Server	*#cs*	*0*	*+1*	*+2*	*+3*	*+4*	*+5*
ML-G callsites							
MySQL	173	163	3	1	1	0	5
Node.js	124	118	6	0	0	0	0
geomean	146	139	4	1	1	0	5
REC-G callsites							
MySQL	278	240	14	11	2	4	7
Node.js	57	55	2	0	0	0	0
geomean	126	115	5	11	2	4	7

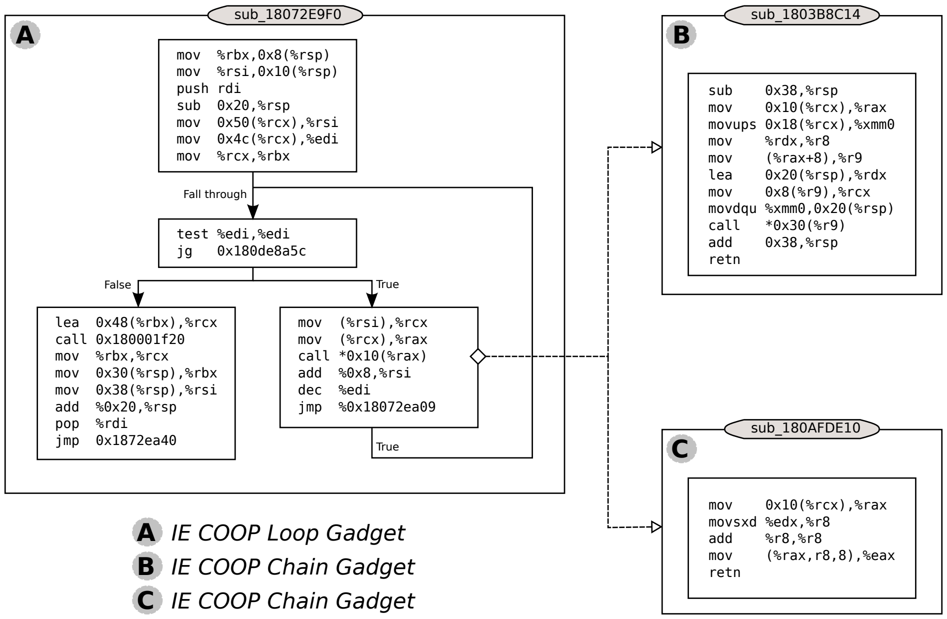
B. Stopping COOP Exploits in Practice

In the following sections, we analyze the published exploits for Internet Explorer (IE), Firefox [26], and Chrome [13] and show how *TypeArmor* stops these attacks.

1) Exploit on 64-bit IE: The original COOP paper presents two exploits against 64-bit IE, both using the main loop gadget ML-G shown in Figure 6 (A). After initialization, the function `sub_18072E9F0` enters a loop and remains looping until `edi` reaches zero. The loop itself (i) loads a (counterfeit) object by setting the first argument (the `this` pointer): `mov (%rsi),rcx`, (ii) prepares and calls a virtual function: `call *0x10(%rax)`, (iii) increases `rsi` to point to the next virtual function: `add $0x8, %rsi`, and (iv) decreases the loop counter: `dec edi`. By controlling memory near `(%rsi)`, the exploit can chain virtual functions and launch the attack[4].

Let us now walk through *TypeArmor*'s callsite analysis. It starts at the basic block that contains the indirect call instruction and concludes that only the first argument (`rcx`) is set in this block. Since there is no conclusive result for the remaining argument registers yet, it moves to the previous block which contains the loop condition (`test %edi,%edi`). As this block does not touch any argument register, it continues by searching for incoming edges to this block. The analysis finds two blocks: the entry block of the function, and the loop block that directly follows the indirect call instruction (ending with `jmp %0x18072ae09`). By following the second edge, we again see no write operations on argument registers, and the analysis must continue by searching for incoming edges to `add %0x8, %rsi`. It is at this moment that *TypeArmor* hits the `call *0x10(%rax)` instruction and can stop its analysis: the call instruction forces the compiler to reset any argument register if it is required by the program later on. So far, *TypeArmor* observed only one argument register to be set and concludes that this callsite sets *at most* one argument.

[4]Note that the Microsoft x64 calling convention is different from the System V AMD64 ABI: only the first four arguments are passed via registers, namely: `rcx`, `rdx`, `r8`, `r9`.



```
mov  %rbx,0x8(%rsp)
mov  %rsi,0x10(%rsp)
push rdi
sub  0x20,%rsp
mov  0x50(%rcx),%rsi
mov  0x4c(%rcx),%edi
mov  %rcx,%rbx
```

```
test %edi,%edi
jg   0x180de8a5c
```

```
lea  0x48(%rbx),%rcx
call 0x180001f20
mov  %rbx,%rcx
mov  0x30(%rsp),%rbx
mov  0x38(%rsp),%rsi
add  %0x20,%rsp
pop  %rdi
jmp  0x1872ea40
```

```
mov  (%rsi),%rcx
mov  (%rcx),%rax
call *0x10(%rax)
add  %0x8,%rsi
dec  %edi
jmp  %0x18072ea09
```

```
sub     0x38,%rsp
mov     0x10(%rcx),%rax
movups  0x18(%rcx),%xmm0
mov     %rdx,%r8
mov     (%rax+8),%r9
lea     0x20(%rsp),%rdx
mov     0x8(%r9),%rcx
movdqu  %xmm0,0x20(%rsp)
call    *0x30(%r9)
add     0x38,%rsp
retn
```

```
mov     0x10(%rcx),%rax
movsxd  %edx,%r8
add     %r8,%r8
mov     (%rax,r8,8),%eax
retn
```

Figure 6. Gadgets used in COOP's 64-bit IE exploit

TypeArmor thus ensures that the indirect callsite in the loop gadget may target only those functions that accept zero or one argument. Looking at the chain of virtual functions, however, we find several *vfgadgets* that use a minimum of two arguments. One such function is `sub_1803B8C14`, illustrated in Figure 6 (B). The first two argument registers `rcx` (via `mov 0x10(%rcx),%rax`), and `rdx` (via `mov %rdx, %r8`) are read-before-write, and *TypeArmor* thus concludes that this is a function that expects *at least* two arguments. *TypeArmor*'s many-to-many map now enforces that the indirect callsite in the loop gadget (that prepares *at most one argument*) is not allowed to call `sub_1803B8C14` (which expects *at least two arguments*). *TypeArmor* thus successfully stops the exploit.

The second COOP exploit against IE relies on the same ML-G, but deploys a different chain of virtual functions. Similar to the first exploit, it uses a virtual function that expects at least two arguments (shown in Figure 6 (C)—`mov 0x10(%rcx),%rax`) and `movsxd %edx,%r8`. Therefore, *TypeArmor* also stops this exploit.

2) Exploit on 64-bit Firefox: We examined COOP's Firefox exploit and found that the ML-G used for the Firefox attack prepares only one argument (the `this` pointer, in `rdi`). Similar to what we observed for the IE exploits, this is correctly inferred by *TypeArmor*'s callsite analysis. Moreover, the gadget chain relies on implicit data flows through argument registers and consists of functions that expect *at least* two arguments, among others. This means that *TypeArmor* successfully stops the Firefox COOP exploit.

3) Exploit on Chrome: In contrast to the ML-G gadgets used by the previous exploits, the improved COOP attack against Google Chrome alternates between two recursive gadgets (REC-G) to chain virtual functions. By analyzing the gadget chain, we find that three consecutive gadgets use `rsi` to pass data. Looking at the `SkComposeShader::contextSize()` REC-G, however, we find that *TypeArmor* identifies that its second indirect call (used to direct control flow to the second REC-G, `blink::XMLHttpRequest::AddEventListener()`), prepares only one argument. This means that *TypeArmor*'s CFC enforcement scrambles data stored in `rsi` and thus breaks the exploit.

Additionally, the first indirect callsite in `SkComposeShader::contextSize()` is non-`void`, meaning that it can only call functions that set `rax`. One of the chained vfgadgets, `TtsControllerImpl::SetPlatformImpl()`, however, is of type `void` and never writes to `rax`. Thus, *TypeArmor*'s CFI mechanism stops this attack as well.

C. Control Jujutsu

The two Control Jujutsu exploits [16] combine data and control-flow diversion attacks: the authors assume a (restricted) memory write to prepare a certain state, followed by overwriting a function pointer. The new function pointer stills targets a function entry, but one that can use the prepared state to give the attacker control over the program [16]. Inspecting the attacks with *TypeArmor* in mind, we can infer that we stop both attacks: (i) the attack against Nginx diverts a non-void callsite in `ngx_output_chain` to target a void function `ngx_execute_proc`, which *TypeArmor* correctly detected as such; (ii) the attack against Apache

HTTPD diverts a callsite in `ap_run_dirwalk_stat` to invoke a target function that does not have its address taken (`piped_log_spawn`), which *TypeArmor* does not allow. Although the authors argue that in this scenario the attack can still succeed by calling `ap_open_piped_log_ex` instead (which wraps `piped_log_spawn`), this is not properly evaluated. By looking at the source code, it is likely that this extra level of indirection corrupts the attacker's prepared state.

D. COOP Extensions

While we demonstrated in Section VI-B how *TypeArmor* stops all published COOP exploits, we now discuss the feasibility of advanced, previously unexplored techniques that could extend COOP.

1) Data Flow in COOP: The original COOP paper presents multiple approaches to pass arguments between vfgadgets, distributed among three classes: (i) data flow using unused argument registers, (ii) data flow using overlapping counterfeit object fields or global variables, and (iii) data flow by relying on arguments actually passed to the callee. Note that the first class specifically targets x86_64, as it mostly uses registers to pass arguments to a function. We refer to this class as *implicit* data flow and for the remaining two as *explicit* data flow.

As the published COOP exploits demonstrate, *implicit* data flow is often key to successful exploitation: in many cases, ML-Gs and REC-Gs prepare only few arguments for the callsite, leaving the attacker with many registers she can use for data flow. Having more registers at her disposal, in turn, increases the probability of finding vfgadgets that implement useful functionality on these registers. One has to make sure, however, that the main-loop (or recursive) gadget does not overwrite said registers in between virtual function calls.

Explicit data flow, on the other hand, is characterized by enabling data flow using *actual* arguments to the vfgadget. Most notably, this also includes the first argument (which, for C++, depicts the object pointer). By overlapping multiple objects of different classes, two vfgadgets may operate on the same (overlapped) object field. This idea can be extended to other arguments as well, which is what COOP uses to enable data flow for their 32-bit IE exploit on Windows x86. In this approach, it uses an initial gadget that always passes the (same) field of the initial counterfeit object to the various vfgadgets. This field can then be used to pass arguments between gadgets and requires vfgadgets to dereference the corresponding argument and read from or write to it. Such field is defined as *argument field* [26].

2) Impact of TypeArmor on Data Flow: TypeArmor effectively prevents implicit data flow. In Section VI-A, we show that static analysis is accurate enough to precisely determine the correct argument count for indirect callsites in many cases. Consequently, CFC scrambles all argument registers that are known to be unused. This prevents implicit data flow by design, both for ML-Gs and REC-Gs.

If *TypeArmor* fails to determine the exact argument count a callsite prepares, an attacker might be able to use this discrepancy to enable data flow. Note, however, that compared to the original COOP setting, she is still severely constrained. First, she does not have as many registers to choose from, which lowers the probability of finding vfgadgets with the desired semantics. Second, CFI is still in place, which significantly reduces the target set. In fact, our manual analysis shows that even for those cases, *TypeArmor* still makes implicit data flow impossible.

Looking at explicit data flows, we distinguish two cases. First, data flows using overlapping object fields, for which we refer to the original COOP paper: it already concludes that these scenarios are difficult to apply in practice. The second case enables a different class of COOP data-flow semantics, which relies on the presence of an argument field. As with the first scenario, however, this is hard to realize in practice since not passing the particular argument may heavily interfere with the program's semantics.

Advanced argument-passing techniques can be tackled by source-level CFI solutions: they have access to type information of the callsite, and can thus enforce a match to types of the callee. In particular, such information can reduce the number of gadgets applicable for data flow via argument fields (object fields that are passed as parameter to a vfgadget by the ML-G). If an analysis determines an argument field to be a pointer, the ML-G's callsite can only target vfgadgets that expect a pointer for the corresponding argument and vice versa. We anticipate that this argument type distinction is also possible at the binary level and consider it as something to explore in future work.

Although we confirm that advanced COOP exploitation is still possible in theory, we stress that a significant reduction of the attack surface at the binary level is possible. In fact, with *TypeArmor* in place, only the really elaborate, but inherently constrained, options for argument passing survive for building working COOP exploits.

E. Pure Data-only Attacks

The Control-Flow Bending (CFB) paper evaluates the general effectiveness of ideal CFI solutions and evidences their limitations against sophisticated CFG-aware attacks [9]. The authors show that CFB attacks against CFI solutions that are complemented by a shadow stack are more difficult, but sometimes still possible.

As any other CFI solution, *TypeArmor* cannot stop pure data-only attacks. However, attacks that use an arbitrary memory write to overwrite a function pointer can still potentially be stopped: if the attacker overwrites a pointer to point to a function that expects more arguments than the original target, or if the new target assumes that certain

Table IV
RUN-TIME PERFORMANCE OVERHEAD FOR SERVER APPLICATIONS. *IC/sec* DEPICTS THE NUMBER OF INDIRECT CALLSITES EXECUTED PER SECOND ON AVERAGE. *CFI* DEPICTS THE OVERHEAD OF *TypeArmor*'S CFI IMPLEMENTATION (CHECKING CALLEE LABELS BEFORE EACH INDIRECT CALL INSTRUCTION). *+CFC* DEPICTS THE SLOWDOWN OF *TypeArmor*'S COMPLETE CONFIGURATION.

Server	*IC/sec*	*CFI*	*+CFC*
Exim	4,574	1.068	1.067
lighttpd	1,425,099	1.116	1.174
Memcached	72,519	1.014	1.017
Nginx	5,084,715	1.132	1.155
OpenSSH	78	1.021	1.013
ProFTPD	542,443	1.007	1.002
Pure-FTPd	17	1.020	1.013
vsftpd	24,024	1.025	1.051
PostgreSQL	18,024,485	1.160	1.205
MySQL	19,693,937	1.239	1.222
Node.js	1,965,955	1.061	1.055
geo-mean	110,157.9	1.076	1.086

callsite arguments that have been scrambled by CFC contain a specific value, the attack will be stopped.

Through personal communication, the CFB authors shared their exploit notes for the presented Apache and Wireshark attacks; two attacks that work even in the presence of a runtime shadow stack and ultimately overwrite a function pointer at some point during the exploit. After analyzing the exploits in depth, we conclude that these truly are pure data-only attacks, and cannot be stopped by *TypeArmor*. It is worth mentioning that even source-level CFI solutions cannot stop these two attacks.

VII. PERFORMANCE

TypeArmor is implemented on Linux for x86_64. The callee and callsite analysis component, outlined in Section IV, is implemented in 5,532 lines of C++ code and depends on the Dyninst v8.2.1 binary analysis framework to disassemble machine code [8]. The runtime component, outlined in Section V, also relies on Dyninst to perform binary instrumentation and consists of 743 lines of code. The prototype supports generic 64-bit ELF binaries as long as they do not emit self-modifying code.

The evaluation testbed is a system equipped with an Intel i5-2400 CPU 3.10GHz and 8GB of RAM. We ran our tests on Ubuntu 14.04 x86_64 running kernel 3.13. We focus our performance evaluation on popular Linux server applications, given that (i) they are widely adopted in the research community for evaluation purposes, (ii) they are popular exploitation targets, and (iii) they naturally contain a relevant number of indirect callsites that can greatly benefit from the protection offered by *TypeArmor*. Specifically, we evaluated *TypeArmor* with three FTP servers (namely, vsftpd v1.1.0, ProFTPD v1.3.3, and Pure-FTPd v1.0.36), two web servers (Nginx v0.8.54 and lighttpd v1.4.28), an SSH server (the OpenSSH Daemon v3.5), an email server (Exim v4.69), two

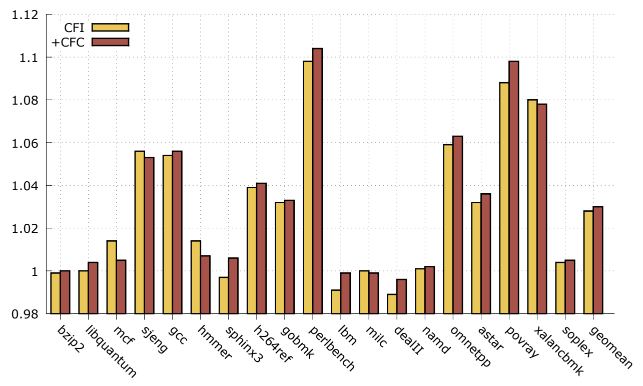


Figure 7. Benchmark run time normalized against the baseline for the SPEC CPU2006 benchmarks.

SQL servers (MySQL v5.1.65 and PostgreSQL v9.0.10), a general-purpose distributed memory caching system (Memcached v1.4.20), and a cross-platform runtime environment for server-side web applications (Node.js 0.12.5, statically compiled with Google's v8 JavaScript engine). Finally, we considered all C and C++ SPEC CPU2006 benchmarks for completeness and direct comparison with prior work.

To benchmark the web servers and Node.js (which we set up to serve a JavaScript page that mimics default webserver behavior), we configured the Apache benchmark [1] to issue 250,000 requests with 10 concurrent connections and 10 requests per connection. To benchmark the FTP servers, we configured the pyftpbench benchmark [4] to open 100 connections and request 100 1 KB-sized files per connection. To benchmark Memcached, we used the memslap benchmark [2]. To benchmark the SQL servers, we configured the Sysbench OLTP benchmark [6] to issue 10,000 transactions using a read-write workload. Finally, to benchmark OpenSSH and Exim, we used the OpenSSH test suite [3] and a homegrown script which repeatedly launches the sendemail program [5], respectively. We configured all applications and benchmarks with their default settings. We ran all the experiments 11 times (checking that the CPUs were fully loaded throughout the tests) and report the median (with marginal standard deviation observed across runs).

To evaluate the impact of *TypeArmor*'s instrumentation on runtime performance, we measured the time to complete the execution of the benchmarks and compared against the baseline. The baseline refers to the original version of the benchmark with no binary instrumentation applied. Table IV details the normalized run time for two configurations. The *CFI* configuration refers to *TypeArmor* solely enforcing forward-edge CFI as outlined in Section V-B. As shown in the table, this configuration introduces a noticeable performance impact (7.6% on average, geometric mean), owing to about half of the applications executing millions of indirect callsites per second. The overhead is compa-

Table V
STATIC ANALYSIS RESULTS AND FORWARD-EDGE TARGET COUNT COMPARISON. THE *Callsites* AND *Callees* GROUPS REPORT STATISTICS ON (I) HOW MANY CALLSITES/CALLEES WERE FOUND, (II) IN HOW MANY CASES OUR STATIC ANALYSIS CORRECTLY IDENTIFIED THE NUMBER OF SET/USED ARGUMENTS, AND (III) THE NUMBER OF CORRECTLY DETECTED NON-VOID CALLSITES AND VOID CALLEES. THE *Targets* GROUP REPORTS THE MEDIAN NUMBER OF TARGETS ACROSS DIFFERENT CFI POLICIES: COARSE-GRAINED CFI (*AT*), AND *TypeArmor*'S *CFI* AND *CFI+CFC*.

	Callsites			**Callees**			**Targets (median)**		
Server	#	*args (perfect%)*	*non-void (correct%)*	#	*args (perfect%)*	*void (correct%)*	*AT*	*CFI*	*+CFC*
Exim	76	65 (85.53)	44 (67.69)	615	495 (80.49)	32 (17.98)	615	41	40
lighttpd	54	47 (87.04)	21 (84.00)	353	311 (88.10)	19 (20.00)	353	50	47
Memcached	48	41 (85.42)	15 (100.00)	236	210 (88.98)	11 (7.91)	236	14	14
Nginx	218	161 (73.85)	155 (90.64)	1,111	869 (78.22)	57 (22.62)	1,111	352	254
OpenSSH	134	130 (97.01)	67 (100.00)	715	625 (87.41)	48 (13.19)	715	32	6
ProFTPD	85	68 (80.00)	62 (93.94)	1,188	1,045 (87.96)	69 (26.74)	1,188	390	376
Pure-FTPd	10	8 (80.00)	3 (50.00)	201	169 (84.08)	0 (0.00)	201	6	4
vsftpd	4	2 (50.00)	1 (100.00)	445	371 (83.37)	29 (12.03)	445	12	12
PostgreSQL	491	392 (79.84)	328 (87.23)	9,312	8,054 (86.49)	494 (15.13)	9,312	2,357	2,304
MySQL	7,532	5,771 (76.62)	4,783 (70.97)	9,961	6,977 (70.04)	1,277 (36.60)	9,961	4,158	3,698
Node.js	2,452	2,113 (86.17)	1,199 (91.39)	34,703	28,698 (82.70)	3,444 (22.26)	34,703	4,804	4,714
geomean	130	103 (79.19)	62 (83.47)	1,270	1,058 (83.26)	98 (17.89)	1,270	141	110

rable to *TypeArmor*'s complete (and default) configuration (*CFI+CFC*), which accounts for *TypeArmor* also clearing unused argument registers at each callsite (8.6% on average, geometric mean). On applications that execute less than a million indirect callsites per second, *TypeArmor* has a marginal performance impact.

To obtain standard and comparable performance results across *TypeArmor*'s configurations, we measured the time to complete the SPEC CPU2006 benchmarks and compared it against the baseline. Again, the baseline refers to the original version of the SPEC2006 benchmarks with no binary instrumentation applied. We present results in Figure 7 and confirm the general behavior observed for the server applications, with an average performance overhead of only 2.4% for *TypeArmor* in a CFI-only configuration and of 2.5% for *TypeArmor* in the default (CFI+CFC) configuration (geometric mean).

Overall, *TypeArmor* imposes a relatively low runtime performance impact on all the test programs considered. This confirms that our lightweight instrumentation is successful in producing a runtime overhead that is comparable to, or even faster than existing binary rewriting-based CFI solutions [34].

VIII. SECURITY ANALYSIS

Common evaluation metrics used to assess the effectiveness of defense mechanisms have been questioned by the community [9]. In this paper, we acknowledge that additional research is required for converging on a more efficient security evaluation system. However, for completeness and comparability with similar works, in this section we evaluate *TypeArmor* using security metrics proposed by other systems.

Table V presents accuracy results for (i) callsite and (ii) callee analysis. In addition, it includes (iii) the median number of legal indirect callsite targets as enforced by existing (binary-level) address-taken-based solutions and *TypeArmor*'s policies (*targets*). To validate *TypeArmor*'s static analysis results—ensuring no underestimation occurs at the callsite and no overestimation is observed at the callee—and to compute the accuracy in detecting return usage and exact number of prepared/consumed arguments, we compared *TypeArmor*'s results against the ground truth generated from source code. For this purpose, we (i) relied on the LLVM framework to compile source code into an intermediate representation (LLVM IR) at different optimization levels, (ii) extracted ground truth numbers (number of arguments prepared for each indirect callsite, number of arguments consumed for each function, and the list of callsites/callees that expect or set a return value), and (iii) lowered LLVM bitcode to machine code (using the same optimization levels) on which we ran *TypeArmor*'s static analysis. Table V reports results for `-O2`, but we observed similar results at other optimization levels. For this experiment, we excluded libraries to ensure a fair comparison across server applications. In addition, we included callee analysis results (second group in Table V) for *all* functions in the program.

Table V shows that the static analysis results are very accurate in identifying the exact number of used arguments (79% for callsites and 83% for callees, respectively, geometric mean). The forward static analysis results are slightly better than those obtained with the backward static analysis, given that the stop condition for the callee analysis is stronger than the one used for callsites. Nonetheless, results are encouraging, given that *TypeArmor* can, overall, compute the exact number of source-level arguments in more than 75% of the cases, while operating entirely at the binary level and in a conservative fashion. Similarly, with a success rate of 83% (geometric mean), results for detecting non-void callsites are also accurate. On the other hand, detecting void functions is much harder: we detect less than 20% of

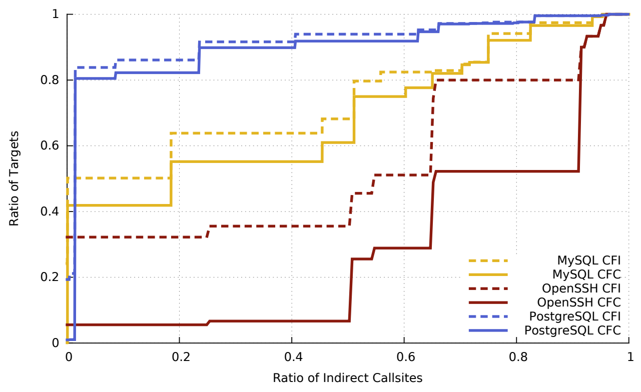


Figure 8. CDF of legal indirect callsite targets enforced by *TypeArmor*'s CFI and CFC policies.

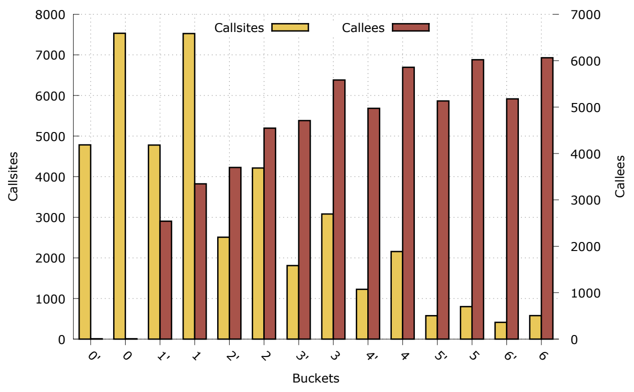


Figure 9. Distribution of CFC buckets for MySQL. A tick (') denotes a bucket containing callsites/callees that expect and set return values.

the actual void callees. This is caused by the fact that **rax** is used as scratch register in many cases, resulting in an underestimation of the number of void functions.

The *targets* column in Table V reflects the static analysis results on the number of legal targets, measuring the strength of CFI and CFC invariants. The *AT* column reports results for existing state-of-the-art binary-level CFI solutions that allow indirect callsites to target any address-taken function [24], [34]. This results in a CFI solution allowing indirect callsites to target all the valid function entry points. The *CFI* and *+CFC* columns report results for *TypeArmor* deployed in a CFI-only configuration and in a full CFI+CFC configuration, respectively.

Table V shows that on average, *TypeArmor* is capable of reducing the number of legal targets by roughly two orders of magnitude (91% reduction on average for CFI+CFC, geometric mean) compared to the conservative address-taken strategy (*AT*) adopted in prior solutions. The results also demonstrate the effectiveness of CFC, which can further reduce the targets allowed by CFI alone (110 vs. 141 targets on average).

For a more accurate view of the invariants enforced by *TypeArmor*, we report a CDF in Figure 8 of legal callsite targets. For clarity, we limit the CDF to CFI and CFC with applications that (i) yield minimal target reduction compared to source-level AT results (PostgreSQL, blue), (ii) contain many indirect callsites and AT functions (MySQL, yellow), and (iii) yield high reduction (OpenSSH, red).

Based on the CDF of Figure 8, we observe that CFC results for each program follow the same trend as CFI. This is inherent to the deployment of the callsite-oriented invariants, with the number of indirect callsites being constant. We observe that results for PostgreSQL, due to the unusual internal structure of the program and the weaker quality of the resulting invariants, are more conservative than other cases, with over 90% of the indirect callsites allowing 80% or more targets. This difference is due to the distribution of the argument count for AT functions: for PostgreSQL, over 85% of the AT functions are detected as consuming at least 0 or 1 argument. This means that as soon as *TypeArmor*'s backward analysis finds that a callsite prepares an additional argument, it must allow all those 85% as a possible target. To make this more concrete, *TypeArmor* concludes that for OpenSSH, only 26 (out of 90) AT functions accept 0 or 1 argument. Encouragingly, other programs exhibit a regular internal structure, resulting in much stronger type-based invariants. For example, we find that results for OpenSSH are impressive: for 90% of all indirect callsites, CFC still yields an almost 50% reduction of the legal targets. Moreover, for 35% of the callsites, *TypeArmor* allows only 7% of all AT functions as valid target.

To further analyze the distribution of possible callees among callsites, Figure 9 depicts a histogram of the different buckets that are enforced by *TypeArmor*'s CFC policy. For each bucket, it shows the number of callees (red) and callsites (yellow) that fall into it. Without return-use information, the System V ABI enables six buckets: callees that take at least 0 arguments, 0 to 1 arguments, 0, 1, or 2 arguments, ..., callees that take any number of arguments. By adding return use information (denoted with a tick ' in Figure 9), the number of buckets is doubled. As an example, consider bucket 3. This bucket contains the callees that expect 0, 1, 2, or 3 arguments, but not those that expect at least 4 arguments or more. On the other hand, it contains callsites that prepare *at most* 6, 5, 4, or 3, arguments, but not 2 or less. Another example is bucket 3', which consists of the same set of only callsites and callees that set and expect a return value.

Figure 9 illustrates the intuitive effectiveness of *TypeArmor*: there is a limited set of callsites (around 500 for MySQL) that are allowed to target any AT function (over 6000), while there are many callsites (7500) that can target only a limited amount of callees (less than 4000). Note that

Table VI
COMPARISON OF *TypeArmor* WITH SOURCE-BASED ADDRESS-TAKEN (AT) ANALYSIS, AND STATE-OF-THE-ART SOURCE-LEVEL CFI DEFENSES (IFCC [29]), IN TERMS OF AVAILABLE TARGETS PER INDIRECT CALLSITE.

	Targets (median)		
Server	*CFI +CFC*	*AT*	*IFCC*
Exim	40	67	3
lighttpd	47	59	6
Memcached	14	14	1
Nginx	254	518	25
OpenSSH	6	90	4
ProFTPD	376	402	3
Pure-FTPd	4	14	0
vsftpd	12	15	1
PostgreSQL	2304	2509	12
MySQL	3698	6097	150
Node.js	4714	7527	341
geomean	110	204	9

since MySQL is a C++ program, and thus `rdi` is often used to hold the `this` pointer, we see almost zero callees in the first two buckets.

Overall, we conclude that *TypeArmor*'s CFI and CFC invariants yield a significant reduction in the number of legal targets at indirect callsites.

A. Comparison with Source-level Techniques

Finally, we compare *TypeArmor* with source-level techniques to assess the strength of the of constraints imposed on indirect callsites. For this purpose, we evaluated a series of well-known programs, and we compared *TypeArmor* with an LLVM-based tool for address-taken (AT) analysis, and state-of-the-art source-level CFI defenses, i.e., IFCC [29]. We present the results in Table VI. As expected, IFCC significantly reduces the available targets of indirect callsites compared to *TypeArmor*. However, note that for certain programs (e.g., OpenSSH) *TypeArmor* performs equally well, although applied at the binary level, and, in *all* cases, *TypeArmor* yields the same or better results than source-based address-taken analysis.

IX. RELATED WORK

Ever since the original CFI proposal by Abadi *et. al.* [7] and the rise of advanced code-reuse attacks [9], [10], [15], [18], [19], [26], [28], there have been several CFI techniques proposed in the literature, targeting both *source* and *binary* compatibility and with different strength of invariants. In this section, we briefly review state-of-the-art CFI solutions *vis-à-vis TypeArmor*.

Binary-level CFI. Realizing binary-level CFI in practice is usually hard, since computing the CFG of a program is an undecidable problem and perfect instrumentation usually incurs overheads. Therefore, there has been research for CFI approximations, realized through coarse-grained CFI [33], [34]. However, these approximations have been demonstrated vulnerable [18]. Lockdown [24] and CFCI [35] attempt to deploy fine-grained CFI schemes, and VTint [32], vfGuard [25], T-VIP [17] focus on protecting just VTables at the binary level. However, it was recently demonstrated that even fine-grained schemes can be bypassed [9], [16] and VTable protections without access to C++ semantics are infeasible [26]. PathArmor shows how recent hardware features can be used to deploy context-sensitive CFI with low overhead [30], but, in absence of forward-edge context-sensitive invariants, COOP attacks are still possible.

In contrast to these solutions, *TypeArmor* enforces strong binary-level invariants based on the number of function arguments, targeting exclusive protection against all these advanced exploitation techniques that bypass fine-grained CFI schemes and VTable protections, at the binary level.

Source-level CFI. Source-level solutions, such as IFCC/VTV [29], SAFEDISPATCH [21], CPI [23], and ShrinkWrap [20] can realize CFI with increased accuracy. In this respect, *TypeArmor* is an attempt to approximate source-level accuracy at the binary level. Although *TypeArmor* is less accurate than such source-level solutions, we argue in this paper that, in the context of sophisticated attacks such as advanced COOP extensions, it is questionable whether additional accuracy that is provided by source-level solutions is required. For most advanced techniques, such as all publicly released COOP exploits, invariants as enforced by *TypeArmor* may be sufficient and effective in practice.

X. CONCLUSION

In this paper, we presented *TypeArmor*, a new detection and containment solution against advanced code-reuse attacks. *TypeArmor* relies on binary-level static analysis to derive both target-oriented and callsite-oriented control-flow invariants and efficiently apply security policies at runtime. In particular, *TypeArmor* relies on target-oriented invariants to enumerate legal callsite targets and *detect* attacks that transfer control to illegal targets (akin to traditional CFI, but with much stronger binary-level invariants). In addition, *TypeArmor* relies on callsite-oriented invariants to invalidate illegal function arguments at each callsite and *contain* attacks that rely on type-unsafe function argument reuse, using a protection technique dubbed Control-Flow Containment (CFC). CFC further improves the quality of our target-oriented invariants, resulting in the strictest binary-level CFI solution to date.

The COOP papers questions whether it is even *possible* to mitigate sophisticated forward-edge attacks using binary-level CFI solutions. *TypeArmor* contrasts these claims with concrete evidence that constructing a strict binary-level CFI solution to counter the most advanced code-reuse attacks in the literature is *possible* and realistic in practice. To substantiate our claims, we demonstrated that *TypeArmor* stops all published COOP exploits.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571, the European Research Council through the ERC Starting Grant No. 640110 (BASTION), and the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing" and NWO CSI-DHS 628.001.021.

References

[1] Apache benchmark. http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] memslap. http://docs.libmemcached.org/bin/memslap.html.

[3] OpenSSH portable regression tests. http://www.dtucker.net/openssh/regress.

[4] pyftpdlib. https://code.google.com/p/pyftpdlib.

[5] SendEmail. http://caspian.dotconf.net/menu/Software/SendEmail.

[6] SysBench. http://sysbench.sourceforge.net.

[7] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *CCS*, 2005.

[8] Andrew R. Bernat and Barton P. Miller. Anywhere, Any-Time Binary Instrumentation. In *PASTE*, 2011.

[9] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX SEC*, 2015.

[10] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX SEC*, 2014.

[11] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, A. R. Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented Programming without Returns. In *CCS*, 2010.

[12] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. In *CCS*, 2015.

[13] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *CCS*, 2015.

[14] Thurston HY Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *ASIACCS*, 2015.

[15] Lucas Davi, A. R. Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX SEC*, 2014.

[16] 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 *CCS*, 2015.

[17] Robert Gawlik and Thorsten Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *ACSAC*, 2014.

[18] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *S&P*, 2014.

[19] 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 *USENIX SEC*, 2014.

[20] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. ShrinkWrap: VTable Protection without Loose Ends. In *ACSAC*, 2015.

[21] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks. In *NDSS*, 2014.

[22] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. 2009.

[23] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *OSDI*, 2014.

[24] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-Grained Control-Flow Integrity through Binary Hardening. In *DIMVA*, 2015.

[25] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in COTS C++ binaries. In *NDSS*, 2015.

[26] 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 *S&P*, 2015.

[27] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *RAID*, 2014.

[28] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *CCS*, 2007.

[29] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX SEC*, 2014.

[30] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *CCS*, 2015.

[31] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *SOSP*, 1993.

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

[33] 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 *S&P*, 2013.

[34] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX SEC*, 2013.

[35] Mingwei Zhang and R. Sekar. Control Flow and Code Integrity for COTS Binaries. In *Proc. ACSAC'15*, 2015.

Appendix

In this appendix, we first formally introduce some key concepts and then we define the invariants that are enforced by *TypeArmor*.

A. Formal Definition of Invariants

Definition A.1. *An indirect callsite cs is said to be of type $\mathbb{T}_{max}$ $(max = 0, 1, 2, \ldots)$ if it* prepares at most max *function arguments (referred to as* actuals*).*

To compute max values for each callsite, *TypeArmor* performs a conservative *backward static analysis* (Section IV-B).

Definition A.2. *A function f has its* address taken *iff the address of f is loaded into memory/registers (referred to as an* address-taken *or* AT *function).*

The set of AT functions determines the superset of targets for unresolved (indirect) forward edges in a program's (interprocedural) control-flow graph. Note that we only focus on forward edges originating from indirect callsites, since *TypeArmor* relies on Dyninst [8] to resolve all the jump tables and corresponding indirect jumps.

Definition A.3. *A function f is said to be of type $\mathbb{T}_{min}$ $(min = 0, 1, 2, \ldots)$ if it* consumes at least min *arguments (referred to as* formals*).*

To compute min values for each function, *TypeArmor* performs a conservative *forward static analysis* (Section IV-A).

Definition A.4. *For a given indirect callsite cs and function f, we define a boolean function $RET(cs, f)$ as follows:*

$$RET(cs, f) = \begin{cases} 0 & \textit{if use\_ret(cs)} \land \textit{is\_void(f);} \\ 1 & \textit{otherwise.} \end{cases}$$

where $use\_ret(cs)$ is true if cs expects a return value (i.e., cs is calling a non-void function) and $is\_void(f)$ is true if function f does not return any value (i.e., f is a void function).

Essentially, $RET()$ enforces the invariant that a non-void callsite can only call non-void functions. Note that this definition still allows void callsites to make calls to non-void functions.

B. Target-oriented invariants for CFI

We now enumerate the key steps *TypeArmor* performs to derive target-oriented invariants and employ them for CFI-style security enforcement:

1) *TypeArmor* scans program and libraries to identify the set $\mathcal{C}$ of indirect callsites with unknown targets.
2) $\forall cs \in \mathcal{C}$, *TypeArmor* performs backward static analysis to determine its type $\mathbb{T}_{max}$. In the following, we denote a callsite cs of a given type $\mathbb{T}_{max}$ as cs_{max}.
3) *TypeArmor* performs static analysis to identify the set of AT functions, generating a superset $\mathcal{F}$ of possible forward-edge targets for each callsite.
4) $\forall f \in \mathcal{F}$, *TypeArmor* performs forward static analysis to determine its type $\mathbb{T}_{min}$. In the following, we denote a function f of a given type $\mathbb{T}_{min}$ as f_{min}.
5) To derive *target-oriented* invariants for each callsite, *TypeArmor* derives a many-to-many mapping $\pi_t : \mathcal{C} \to 2^{\mathcal{F}}$ such that $\pi_t(cs_{max}) = \{f_{min}\colon min \leq max \ \land \ RET(cs_{max}, f_{min}) = 1\}$.
   Note that a many-to-many type-based mapping is necessary due to inability to precisely reconstruct one-to-one function signatures [25]. In binaries, it is perfectly legal for an indirect callsite to target a function with a number of formals lower than the number of actuals prepared at the callsite.
6) *TypeArmor*'s runtime component instruments each $cs \in \mathcal{C}$ and $f \in \mathcal{F}$ according to the mapping π_t to enforce CFI during the execution. That is, *detecting* violations whenever an edge originating from a given callsite cs_{max} targets a function f_{min}, such that $f_{min} \notin \pi_t(cs_{max})$.

C. Callsite-oriented invariants for CFC

TypeArmor complements its target-oriented invariants with stronger callsite-oriented invariants that are employed by CFC. To derive those invariants for CFC-style security enforcement, *TypeArmor* performs the following steps:

1) *TypeArmor* identifies the set $\mathcal{C}$ of indirect callsites and, $\forall cs \in \mathcal{C}$, determines its type $\mathbb{T}_{max}$.
2) To derive *callsite-oriented* invariants for each callsite, *TypeArmor* derives a one-to-one mapping $\pi_c : \mathcal{C} \to \mathbb{T}$ such that $\pi(cs_{max}) = \mathbb{T}_{max}$.
3) *TypeArmor*'s runtime component instruments each $cs \in \mathcal{C}$ according to the mapping π_c to enforce CFC during the execution. That is, setting all the actuals $k > max$ to random values and *containing* illegal edges which originate from a given callsite cs_{max} and target a function with more than max formals.