TypeShield: Practical Forward Call Based Attacks Protection

Anonymous Author(s)

ABSTRACT

Applications aiming for high performance and availability draw on several features in the C/C++ programming language. A key building block are virtual functions, which facilitate late binding, and thereby facilitate runtime polymorphism. However, practice-driven and academic research have identified an alarmingly high number of virtual pointer corruption vulnerabilities which undercut security in significant ways and are still in need of a thorough solution approach.

We contribute to this research area by proposing TypeShield, a binary runtime virtual pointer protection tool which is based on instrumentation of program executables at load time. TypeShield applies a novel runtime type and function parameter counter technique in order to overcome the limitations of available approaches and to efficiently verify dynamic dispatches during runtime. To enhance practical applicability, TypeShield can be automatically and easily used in conjunction with legacy applications or where source code is missing to harden binaries. We have applied TypeShield to web servers, FTP servers and the SPEC CPU2006 benchmark and were able to efficiently and with low performance overhead protect these applications from forward indirect edge corruptions based on virtual pointers. Further, in a direct comparison with the state-of-the-art tool, TypeShield achieves higher caller/caller matching (i.e., precision), while maintaining a more favorable runtime overhead. Focusing the evaluation on target reduction techniques, we can demonstrate that our approach achieves a notable additional reduction of the possible calltargets per callsite of up to 20% associated with an overall reduction of about 9% in comparison to other state-of-the-art parameter-only count-based techniques.

KEYWORDS

C++ object dispatch, indirect call, forward edge, code reuse attack

1 INTRODUCTION

The object-oriented programming (OOP) paradigm is *de facto standard* concept for developing large, complex and efficient systems because it facilitates inheritance for objects which seem to be at first not related to each other; this in turn facilitates better code reuse, software maintenance and software design. There are many programing languages which support OOP concepts, however the C++ programing language is the most used programming language for systems where runtime performance and reliability are the main goal.

An important C++ OOP convention is the object calling convention when for instance virtual functions are called. Virtual functions are an important concept which facilitates late binding and allows the programmer to overwrite a virtual function of the base-class with his own implementation. In fact, in order to implement virtual functions, the compiler needs to generate a table (*i.e.,* virtual table meta-data structure) of all virtual functions for each class containing them and provide to each instance of such a class (*i.e.,* object)

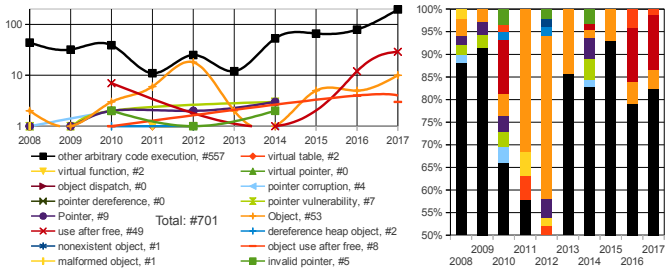

Figure 1: # (left Fig.), % (right Fig.) of arbitrary code executions (ACE) reports related (all colors expect black) to pointer or virtual table (vptr/vtbl) corruption (see bag of words at the bottom of left Fig.)* reported by US NVD for the past 10 years [20]. In black are the ACE unrelated reports. X axis is years (left & right) and Y axis is number of reports in logarithmic scale (left) and distribution in % of the same reports (right). As of May'17, NVD reports in total 701 ACEs from which 143 are the cause of vptr/vtable corruption (see * above) which are exploited by highjacking forward indirect calls. We manually inspected all 701 reports and can confirm that 143 are based on *. The vulnerabilities were reported in applications such as Google's Chrome & V8 JS eng.; Mozilla Firefox; Microsoft's IE 10, Edge & Chakra JS eng.; & iOS/MacOS apps.

a pointer (*i.e.,* virtual pointer) to the aforementioned table. While this allows more flexible code to be built—through late binding—it is simplistically implemented with no security mechanisms at in place.

The performance benefit of late binding comes with high security implications (*i.e.,* 701 arbitrary code executions [20] reported by US NIST NVD, see Figure 1). First, dangling object pointers lead to undefined behavior—as specified by the C++ language standard N4618 [5]—which can be exploited to point into illegal (*i.e.,* not previously intended) virtual tables. Second, trough memory corruptions (*e.g.,* buffer/integer overflows) the virtual pointer of an object making a call to a virtual function can be corrupted to point into: *1)* illegal virtual tables, *2)* newly inserted virtual tables, or *3)* overwritten virtual table entries such that advanced Code-Reuse Attacks (CRAs) as the advanced COOP [24] attack and its extensions [2, 12, 12, 18, 19] become easily doable. This type of attack can bypass most of the to date CFI-based enforcement policies, since: *1)* it does not exploit indirect backward edges (*i.e.,* return edges) but rather *2)* it exploits the forward indirect control flow transfers imprecision which can not be statically upfront determined since alias analysis is undecidable [23] in program binaries.

To avoid object dispatch corruptions Control-Flow Integrity (CFI) [6, 7] can be successfully used. CFI is one of the most used techniques for securing indirect control flow transfers inside programs by usually adding runtime checks before each indirect call site.

Source code based tools usually insert runtime checks during the compilation of the program such as SafeDispatch [15], ShrinkWrap [14] and IFCC/VTV [25]. Other tools modify and reorder the contents of the virtual table layout such as VTI [9] in order to derive efficient range checks on each object dispatch during runtime. Nevertheless, to the best of our knowledge due to runtime performance issues only IFCC/VTV [25] is currently in production available.

Binary based tools typically enforce imprecise forward-edge CFI policies, often allowing control transfers from any valid call site to any valid referenced entry point *e.g.,* binCFI [29, 30]. 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 [21].

TypeArmor [26] implements a fine grained forward edge CFI policy based on parameter count for binaries. It calculates invariants for call targets and indirect call sites based on the number of parameters they use by leveraging static analysis of the binary, which then is patched to enforce those invariants during runtime. The main shortcoming of TypeArmor is that it has low precision w.r.t. to the number of call targets allowed per call site (see **??** for more details).

These source code tools offer a certain degree of protection when code is provided, however the above mentioned binary tools offer limited or no protection due to an in first place imprecise calltarget set per callsite.

In this paper, we present TypeShield, a runtime binary-level illegitimate forward calls filtering tool that is based on an improved forward-edge fine-grained CFI policy compared to previous work [12, 26]. TypeShield analyzes only 64-bit binaries and only function parameters which are passed with the help of registers. This means that based on the used ABI, TypeShield is able to track 4 or 6 arguments for the Microsoft's x64-bit calling convention or System V ABI, respectively. Similarly to TypeArmor we do not take into consideration floating-point arguments passed via xmm registers; which we want to address in future work. However, as we will demonstrate in the evaluation section, this will provide us enough information to more be precisely than TypeArmor when stopping several state-of-the-art CRAs.

More precisely, the analysis performed by TypeShield: *1)* uses for each function parameter its register wideness (*i.e.,* ABI dependent) in order to map calltargets per callsites, *2)* uses an address taken (AT) analysis similar to [26] for all calltargets, and *3)* compares individually parameters of callsites and calltargets in order to check if an indirect call transfer is acceptable or not, thus this providing a more fine-grained calltarget set per callsite than other state-of-the-art tools. TypeShield is based on a use-def callees analysis to approximated the function prototypes, and liveness analysis at indirect callsites to approximate callsite signatures. This efficiently leads to a more precise CFG of the binary program in question, which can be used also by other systems in order to gain a more precise CFG on which to enforce other types of CFI related policies. TypeShield incorporates an improved protection policy which is based on the insight that if the binary adheres to the standard calling convention for indirect calls, undefined arguments at the call site are not used by any callee by design. This further helps to reduce the possible target set of callees for each callsite. TypeShield relies on a more precise than TypeArmor construction of both the callee parameter types and call site signatures. TypeShield uses automatically inferred parameter types which are later used into the classification of matching call sites and call targets. This helps to obtain more precise callee target sets for each caller as the TypeArmor. TypeShield compared to TypeArmor uses different analysis strategies for basic block merging. Furthermore, TypeShield disallows an indirect call transfer that prepares fewer arguments than the target callee consumes and where the types of the arguments provided are not super types of the arguments expected at the target. It then uses this information to enforce that each call site targets only a strict call target set. TypeShield takes the binary of a program as input and it automatically instrument it in order to detect illegitimate indirect calls at runtime. More precisely, TypeShield achieves three goals.

Precision. TypeShield employs a more precise analysis than TypeArmor in order to reduce the call target set for each call site. Our evaluation shows that TypeShield incurs X% precision w.r.t. TypeArmor on the same programs.

Performance. TypeShield employs runtime policy optimization techniques to further reduce the runtime overheads Our evaluation shows that TypeShield imposes up to X% and X% overheads for performance-intensive benchmarks on the SPEC CPU2006 benchamrks and the webser applications, respectively. On the contrary, TypeArmor is X% slower than TypeShield on the x Program.

Scope. TypeShield can detect forbidden indirect calls and as such it can protect similarly as vTrust [27] against virtual table injection, corruption and reuse attacks. As such TypeShield can serve as a platform for developing other types of defenses for different types of attacks.

In summary, we make the following contributions:

- **Security analysis of forward indirect calls.** We analyzed the usage of illegitimate indirect forward calls in detail, thus providing security researchers and practitioners a better understanding of this emerging threat.
- **Illegitimate indirect calls detection tool.** We designed and implemented TypeShield, a general, automated, and easy to deploy tool that can be applied to C/C++ binaries in order to detect and mitigate illegitimate forward indirect calls during runtime.
- **Experiments.** We demonstrate trough extensive experiments that our precise binary-level CFI strategy can mitigate advanced code reuse attacks in absence of C++ semantics. For example TypeShield can protect against the COOP attack and its variations.

2 FORBIDDEN FORWARD CALLS EXPOSED

In this section, we present a brief overview of the concept of C++-based polymorphism in **??** and how indirect calls can be checked in practice in **??**. In §2.2 we present a forward edge function parameter count based policy [26] and in **??** we present security implications of indirect calls. Finally, in **??** we present an imprecise parameter count based policy, and in **??** we present a real COOP attack example.

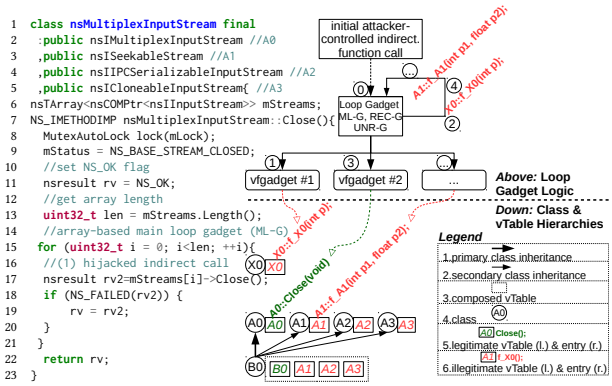

Figure 2: Description of how a counterfeit object-oriented programming main loop gadget (ML-G) works.

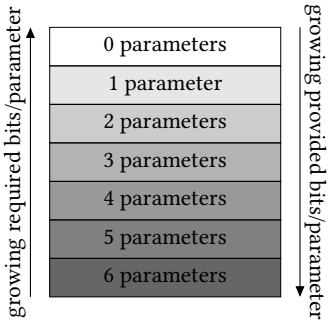

Figure 3: *Count* policy classification schema for callsites and calltargets.

2.1 Exploiting Polymorphism Weaknesses

Figure 2 depicts a C++ code example where it is illustrated how a COOP loop gadget (*i.e.,* ML-G, REC-G, UNR-G, see [12]) works. The vfgadget (1) can be exploited in several ways, see *x), y) and z)* above. The indirect callsite (Figure 2 line 17) can be exploited to call by passing a varying number of parameters and types on each object contained in the array a different vTable entry contained in the: *1)* class hierarchy (overall, whole program), *2)* class hierarchy (partial, only legitimate for this callsite), *3)* vTable hierarchy (overall, whole program), *4)* vTable hierarchy (partial, only legitimate for this callsite), *5)* vTable hierarchy and/or class hierarchy (partial, only legitimate for this callsite), and *6)* vTable hierarchy and/or class hierarchy (overall, whole program). There is no language semantics—such as cast checks—in C++ for vCall sites dispatch checking and as consequence the loop gadget indicated in Figure 2 can basically call all around in the class and vTable hierarchy by not being constrained by any build in check during runtime. The attacker corrupts an indirect function call, (1), next she invokes gadgets, (1) and (3), through the calls, (2) and (4), contained in the loop. As it can be observed in Figure 2 she can invoke from the same callsite legitimate functions residing in the vTable inheritance path (*i.e.,* this type of information is usually very hard to recuperate from executables) for this particular callsite, indicated with green color vTable entries. However, a real COOP attack invokes illegitimate vTable entries residing in the whole initial program hierarchy (or the extended one) with less or no relationship to the initial callsite, indicated with red color vTable entries.

2.2 *Count* Policy

What we call the *count* policy is essentially the policy introduced by TypeArmor [26]. The basic idea revolves around classifying calltargets by the number of parameters they provide and callsites by the number of parameters they require. The schema to match this is based on the fact that we have calltargets requiring parameters and the callsites providing them as depicted in Figure 3.

Furthermore, generating 100% precise measurements for such classification with binaries as the only source of information is rather difficult. Therefore, over-estimations of parameter count for callsites and underestimations of the parameter count for calltargets is deemed acceptable. This classification is based on the general purpose registers that the call convention of the current ABI—in this case the SystemV ABI—designates as parameter registers. Furthermore, we completely ignore floating point registers or multi-integer registers. The core of the *count* policy is now to allow any callsite cs, which provides c_{cs} parameters, to call any calltarget ct, which requires c_{ct} parameters, iff $c_{ct} \leq c_{cs}$ holds. However, the main problem is that while there is a significant restriction of calltargets for the lower callsites, the restriction capability drops rather rapidly when reaching higher parameter counts, with callsites that use 6 or more parameters being able to call all possible calltargets: $\forall cs_1, cs_2 . c_{cs_1} \leq c_{cs_2} \implies \|\{ct \in \mathcal{F} | c_{ct} \leq c_{cs_1}\}\| \leq \|\{ct \in \mathcal{F} | c_{ct} \leq c_{cs_2}\}\|$.

One possible remedy would be the ability to introduce an upper bound for the classification deviation of parameter counts, however as of now, this does not seem feasible with current technology. Another possibility would be the overall reduction of callsites, which can access the same set of calltargets, a route we will explore within this work.

3 OVERVIEW

In this section, we present a brief overview on the considered adversary model in §3.1 and depict the invariants for calltargets and callsites in §3.2. Finally, in §3.3 we present our function parameter type aware policy and give a formal description of it by relating it to [26], and in §3.4 we highlight the impact of our policy on COOP.

3.1 Adversary Model and Assumptions

We largely use the same threat model and the same basic assumptions as described in the TypeArmor paper [26], meaning that our attacker has read and write access to the data sections of the attacked binary. We also assume that the protected binary does not contain self modifying code, handcrafted assembly or any kind of obfuscation. We also consider pages to be either writable or executable but not both at the same time. We assume that our attacker has the ability to execute a memory corruption to hijack the programs control flow and that a solution for backward CFI is in place.

3.2 Invariants for Calltargets and Callsites

Advanced code reuse attacks change the calltargets that are invoked within indirect callsites. As standard CFI solutions can hardly restrict these, TypeArmor proposed using two base invariants: *1)* indirect callsites provide a number of parameters (*i.e.,* possibly overestimated compared to source), and *2)* calltargets require a minimum number of parameters (*i.e.,* possibly underestimated compared to source). The idea is that a callsite might only call functions that do not require more parameters than provided by the callsite. To compute the necessary information, TypeArmor uses a modified version of forward liveness analysis for call-targets and backward reaching definitions analysis for callsites.

3.3 TypeShield Policy Mechanism

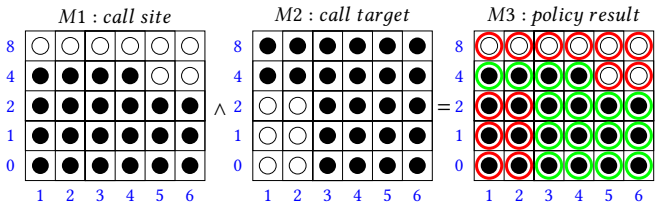

Figure 4: TypeShield's parameter type and count policy. The X and Y axis of matrices $M1, M2$ and $M3$ represent function parameter count and bit-widths in bytes, respectively. Note that our type policy performs an $\wedge$ (*i.e.,* logical and) operation between each entry in $M1_{i,j}$ and $M2_{i,j}$ where i and j are column and row indexes. If two black filled circles located in $M1 \wedge M2$ overlap on positions $M1_i = M2_i \wedge M1_j = M2_j$ than we have a match. Green circles indicate a match whereas red circles indicate a mismatch in $M3$. If at least one match is present on each of the columns of $M3$ than the indirect call transfer will be allowed by our policy, otherwise not. Note that in this example the indirect call transfer will be allowed.

Figure 4 depicts the behavior of our type based policy when the callsite provides 6 parameters $pcs1, \ldots, pcs6$ having following bit wideness $pcs1$: 4-byte, $pcs2$: 4-bye, $pcs3$: 4-byte, $pcs4$: 8-byte, $pcs5$: 2-byte, $pcs6$: 2-byte, and the calltarget is expecting 6 parameters $pct1, \ldots, pct6$ having following bit wideness $pct1$: 4-byte, $pct2$: 4-bye, $pct3$: 0-byte, $pct4$: 0-byte, $pct5$: 0-byte, $pct6$: 0-byte of the expected parameters. TypeShield's type policy is defined as follows.

Definition 3.1. Let A be a call target ct_A and B a call site cs_B than: $ct_A \subseteq cs_B \iff \forall i \subseteq [1,6]$, wideness(parameter($A$)[i]) $\leq$ wideness(parameter(B)[i]).

Whereas the policy of TypeArmor is the following.

Definition 3.2. Let A be a call target ct_A and B a call site cs_B than: $ct_A \subseteq cs_B \iff \forall i \subseteq [1,6]$, count(parameter($A$)) $\leq$ count(parameter(B)).

From Definitions (3.1) and (3.2) it can be observed that the first policy is more fine-grained than the second one since it performs checks for each parameter index in part.

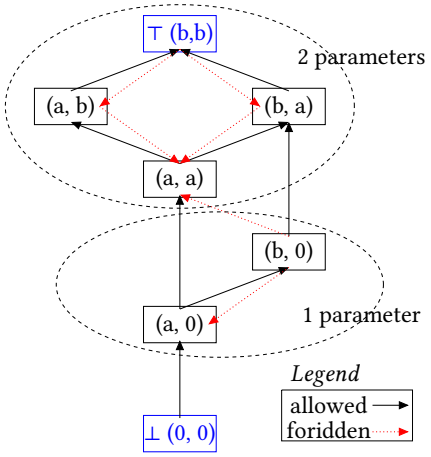

Figure 5: Transition based lattice between call targets and call sites, $a \wedge b \in \{0-bit, 8-bit, 16-bit, 32-bit, 64-bit\}$ and the two function parameters (for brevity) having $\{0-byte, 1-byte, 2-byte, 4-byte, 8-byte\}$ register wideness. TypeShield allows a transition from $a \rightarrow b \; iff \; a_i \leq b_i$ where $i \in [1,2]$. Note that $\top$ and $\bot$ represent the top and bottom elements of the lattice, respectively. An arrow represents an indirect control flow transfer from a callsite to a calltarget. The given lattice contains in total 8 black colored (legal) and 6 red colored (illegal) indirect control flow transitions. Note that [26] would allow all 14 indirect control flow transfers whereas TypeShield would allow the black colored and forbid the red colored transitions.

3.4 TypeShield Impact on COOP

Figure 5 represents the a sub-part of the total indirect transfers space in any given C/C++ program. In case a CFI policy schema is based only on parameter count with callsite overestimation and calltarget subestimation it is possible that a callsite can use any calltarget as long as the number of parameter provided and required are fullfiling the policy, even if the parameter types do not match (*i.e.,* imagine 8-bit values provided by the callsite but 64-bit values required by the calltarget). Such a parameter count based policy is *blind* and would allow any call transfer inside the lattice space presented in Figure 5 and as such the calltarget set per callsite would be too permissive.

In order to effectively deal with this situation we extend the above presented parameter count based policy in order to be able to deal with function parameter types as well.We introduce the following policy rules: *1)* indirect callsites provide a maximum wideness to each parameter, and *2)* calltargets require a minimum wideness for each parameter. Note that for both rules the minimum and maximum wideness for each function parameter is possibly underestimated compared to the source code of the program with which we also compare in §6. Note, that the number of provided parameters must be no lower than the requirement the number of consumed parameters. Finally, our approach is more fine-grained by considering parameter wideness and as such the allowed calltarget lattice space is considerably reduced.

4 DESIGN

In this section, we cover the design of TypeShield. We first present theory and definitions for our instructions analysis based on register states. §4.1. Then we present the details of our new *type* policy in §4.2. Finally we present the design of our calltarget analysis in §4.3 and the design of our callsite analysis in §4.4 .

4.1 Analysis of Register-States

Instead of symbol based data-flow analysis, our approach is register state based. Therefore we need to adapt the usual definitions.

The set INSTR describes all possible instructions that can occur within the executable section of a binary. In our case this is based on the instruction set for x86-64 processors.
An instruction $i \in \texttt{INSTR}$ can non-exclusively perform two kinds of operations on any number of existing registers:[1]
1) Read n-bit from the register with $n \in \{64, 32, 16, 8\}$
2) Write n-bit to the register with $n \in \{64, 32, 16, 8\}$
We descibe the possible change within one register as $\delta \in \Delta$ with $\Delta = \{w64, w32, w16, w8, 0\} \times \{r64, r32, r16, r8, 0\}$. [2]
SystemV ABI specifies 16 general purpose integer registers. Therefore we represent the change occurring at the processor level as $\delta_p \in \Delta^{16}$. We calculate this change for each instruction $i \in \texttt{INSTR}$ via the function $decode : \texttt{INSTR} \mapsto \Delta^{16}$.

4.2 *Type* Policy

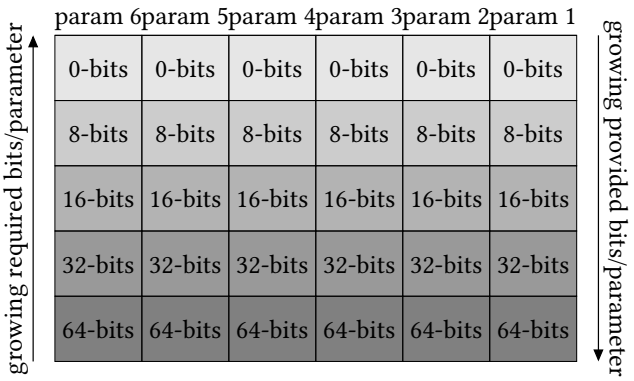

Figure 6: The *type* policy schema for callsites and calltargets. As is demonstrated here, when requiring wideness, one starts at the bottom and grows to the top, as it is always possible to accept more than one requires. The reverse is true for providing, as it is possible to accept less than provided.

As shown in Figure 6, our idea is to not simply classify callsites and calltargets based on the number of parameters they provide or request, but als on the parameter type. To simplify our approach we use the wideness of the type and do not infer the actual type.

[1]There are registers that can directly access the higher 8-bit of the lower 16-bit. For our purpose we register this access as a 16-bit access.
[2]Note that 0 signals the absence of either a write or read access and (0, 0) signals the absence of both. Furthermore, wn or rn with $n \in \{64, 32, 16, 8\}$ implies all wm or rm with $m \in \{64, 32, 16, 8\}$ and $m < n$ (*e.g.,* $r64$ implies $r32$). Note that we exclude 0, as it means the absence of any access.

As previously mentioned there are 4 types of reading and writing access each. Therefore our set of possible types for parameters is $\texttt{TYPE} = \{64, 32, 16, 8, 0\}$; 0 models the absence of a parameter. Since SystemV ABI specifies 6 registers as parameter holding registers, we classify our callsites and calltargets into $\texttt{TYPE}^6$.
Similar to the policy of TypeArmour, we allow overestimations of callsites and underestimations of calltargets, however on the level of types. Therefore for a callsite cs to be able to call the calltarget ct, if for each parameter of ct the corresponding parameter of cs is not smaller when comparing the wideness.
This results in a finer-grained policy further restricting the possible pool of calltargets for each callsite.

4.3 Calltarget Analysis

For our policy we need to classify our calltargets according to the parameters they provide. Underestimations are allowed, however overestimations shall not be permitted. For this purpose we employ a customizable modified liveness analysis algorithm, which we will show first. We then present our versions for a *count* and a*type* based policy. Furthermore, we need to be aware of certain corner cases, which we will discuss at the end.

Liveness Analysis A variable is alive before the execution of an instruction, if at least one of the originating paths performs a read access before any write access on that variable. If applied to a function, this calculates essentially the variables that need to be alive at the beginning, which in essense are its parameters. We based Algorithm 1 on the liveness analysis algorithm in Khedker *et al.* [16], which essentially is a depth first traversal of blocks. For customization we rely on the implementation of several functions($\mathcal{S}^{\mathcal{L}}$ is the set of possible register states depending on the specific liveness implementation):

`merge_v` : $\mathcal{S}^{\mathcal{L}} \times \mathcal{S}^{\mathcal{L}} \mapsto \mathcal{S}^{\mathcal{L}}$, which describes how to merge a set of states resulting from several paths.

`merge_h` : $\mathcal{P}(\mathcal{S}^{\mathcal{L}}) \mapsto \mathcal{S}^{\mathcal{L}}$, which describes how to merge the current state with the following state change.

`analyze_instr` : $\texttt{INSTR} \mapsto \mathcal{S}^{\mathcal{L}}$, which calculates the state change that occurs due to the given instruction

`succ` : $\texttt{INSTR}^* \mapsto \mathcal{P}(\texttt{INSTR}^*)$, which calculates the successors of the given block.

In our specific case, the function `analyze_instr` needs to also handle non jump and non fallthrough successors, as these are not handled by DynInst. Essentially there are four relevant cases: 1) If the current instruction is an indirect call or a direct call and the chosen implementation should not follow calls, then return a state where all registers are considered to be written before read. 2) If the current instruction is a direct call and the chosen implementation should follow calls, then we start an analysis of the target function an return its result. 3) If the instruction is a constant write (e.g., xor of two registers), then we remove the read portion before we return the decoded state. 4) In any other case, we simply return the decoded state.
This leaves us with the two undefined merge functions and the undefined liveness state $\mathcal{S}^{\mathcal{L}}$. In the following two paragraphs we will present two implementation variants: first similar to TypeArmour

Algorithm 1: Basic block liveness analysis.

Input : block : INSTR*
Output : $\mathcal{S}^{\mathcal{L}}$

```
1  Function analyze (block : INSTR*) : S^L is
2      state = Bl ;                          ▷ Initialize the state
3      foreach inst ∈ block do
4          state' = analyze_instr(inst) ; ▷ Calculate changes
5          state = merge_h(state, state') ;      ▷ Merge changes
6      end
7      states = {} ;                  ▷ Set of surccessor states
8      blocks = succ(block) ;                 ▷ Get successors
9      foreach block' ∈ blocks do
10         state' =  analyze(block') ;   ▷ Analyze successor
11         states = states ∪ { state' } ; ▷ Add successor states
12     end
13     state' = merge_h (states) ;   ▷ Merge successor states
14     return merge_v(state, state') ; ▷ Merge to final state
15 end
```

a *count* based policy and second our *type* based policy.

Required Parameter Count To implement the *count* policy, we only need a coarse representation of the state of one register, thus we use the same representation as TypeArmor: 1) W represents write before read access, 2) R represents read before write access, and 3) C represents the absence of access. This gives us the $S^{\mathcal{L}} = \{C, R, W\}$ as register state, which translates to the register super state $\mathcal{S}^{\mathcal{L}} = (S^{\mathcal{L}})^{16}$.
We implement `merge_v` in such a way that a state within a super-state is only updated if the corresponding register has yet to be accesses, as represented by C. Our reasoning is that the first access is the relevant one to determine read before write.

TODO `merge_h` (depends on numbers)

The index of highest parameter register based on the used call convention that has the state R is considered to be the number of parameters a function at least requires to be prepared by a callsite.

Required Parameter Wideness. To implement the *type* policy, we need a finer representation of the state of one register: *1)* W represents write before read access, *2)* $r8, r16, r32, r64$ represents read before write access with 8-, 16-, 32-, 64-bit wideness, and *3)* C represents the absence of access.

This gives us the following register state $S^{\mathcal{L}} = \{C, r8, r16, r32, r64, W\}$ which translates to the register super state $\mathcal{S}^{\mathcal{L}} = (S^{\mathcal{L}})^{16}$. As there could be more than one read of a register before it is written, we might be interested in more than just the first occurrence of a write or read on a path. To allow this we allow our merge operations to also return the value RW, which represents the existence of both read and write access and then can use W as an end marker of sorts. Therefore, our vertical merge operator conceptually intersects all read accesses along a path until the first write occurs ($merge\_v^i$). In any other case it behaves like the previously mentioned vertical merge function. Our horizontal merge($merge\_h$) function is again a simple pairwise combination of the given set of states, which are then combined with a union like operator with W preceding WR preceding R preceding C. Unless one side is W, read accesses are combined in such a way that always the higher one is chosen.

Variadic Functions. Variadic functions are special functions in C/C++ that have a basic set of parameters, which they always require and a variadic set of parameters, which as the name suggests may vary. A prominent example of this would be the *printf* function, which is used to output text to *stdout*.

The problem with these functions is that to allow for easier processing of parameters usually all potential variadic parameters are moved into a contiguous block of memory Our analysis interprets that as a read access on all parameters and we arrive at a problematic overestimation.

```
00000000004222f0 <make_cmd>:
 4222f0:push   %r15
 4222f2:push   %r14
 4222f4:push   %rbx
 4222f5:sub    $0xd0,%rsp
 4222fc:mov    %esi,%r15d
 4222ff:mov    %rdi,%\begin{figure}[!h]
 422302:test   %al,%al
 422304:je     42233d <make_cmd+0x4d>
 422306:movaps %xmm0,0x50(%rsp)
 42230b:movaps %xmm1,0x60(%rsp)
 422310:movaps %xmm2,0x70(%rsp)
 422315:movaps %xmm3,0x80(%rsp)
 42231d:movaps %xmm4,0x90(%rsp)
 422325:movaps %xmm5,0xa0(%rsp)
 42232d:movaps %xmm6,0xb0(%rsp)
 422335:movaps %xmm7,0xc0(%rsp)
 42233d:mov    %r9,0x48(%rsp)
 422342:mov    %r8,0x40(%rsp)
 422347:mov    %rcx,0x38(%rsp)
 42234c:mov    %rdx,0x30(%rsp)
 422351:mov    $0x50,%esi
 422356:mov    %r14,%rdi
 422359:callq  409430 <pcalloc>
```

Figure 7: ASM code of the `make_cmd` function with optimize level O2, which has a variadic parameter list.

Our solution to this problem is to find these spurious reads and ignore them. A compiler will implement this type of operation very similar for all cases, thus we can achieve this using the following steps: *1)* we look for what we call the xmm-passthrough block, which entirely consist of moving the values of registers `xmm0` to `xmm7` into contiguous memory, *2)* we look at the predecessor of the xmm-passthrough block, which we call the entry block (in our case basic block Check if the successors of the entry block consist of the xmm-passthrough block and the successor of the xmm-passthrough block, which we call the param-passthrough block, and *3)* We look at the param-passthrough block and set all instructions that move the value of a parameter register into memory to be ignored.

Ignoring Reads. When one instruction writes and reads a register at the same time we give the read access precedence, however there are exceptions (also mentioned in TypeArmor, however we expand slightly on that): *1)* `xor %rax, %rax` is the first obvious scenario, as it will always result in `%rax` holding the value 0, *2)* `sub %rax, %rax` is probably the next scenario, as it results also in `%rax` also holding the value 0, and *3)* `sbb %rax, %rax` is also relevant, however it will not result in a constant value and based

on the current state might either result in `%rax` holding the value 0 or 1.

Algorithm 2: Basic block reaching definition analysis.

Input : basic block
Output : $\mathcal{S}^{\mathcal{R}}$

```
1  Function analyze(block : BasicBlock) : S^R is
2      state = Bl ;                          ▷ Some comment
3      foreach inst ∈ reversed(block) do
4          state' = analyze_instr(inst) ;    ▷ Some comment
5          state = merge_v(state, state') ;  ▷ Some comment
6      end
7      states = {} ;                         ▷ Some comment
8      blocks = pred(block) ;                ▷ Some comment
9      foreach block' ∈ blocks do
10         state' = analyze(block') ;        ▷ Some comment
11         states = states ∪ { state' } ;     ▷ Some comment
12     end
13     state' = merge_h (states) ;           ▷ Some comment
14     return merge_v(state, state') ;       ▷ Some comment
15 end
```

4.4 Callsite Analysis

For either *count* or *type* policy to work, we need to arrive at an overestimation of the provided parameters by any indirect callsite existing within the targeted binary. We will employ a modified version of reaching analysis that tracks registers instead of variables to generate the needed overestimation. As our algorithm will be customizable, we look at the required merge functions to implement *count* and *type* policy.

Reaching Definitions Theory. An assignment of a value to a variable is a reaching definition at the end of a block n, if that definition is present within at least one path from start to the end of the block n without being overwritten by another value assignment to the same variable. We employ reaching definitions analysis, because we are looking for the parameters a callsite provides. This essentially requires the last known set of definitions that reach the actual call instruction within the parameter registers.

The book [16] defines reaching definition analysis on blocks, which we use to arrive at algorithm depicted in Algorithm 2 to compute the liveness state at the start of a basic block. We apply the reaching analysis at each indirect callsite directly before each call instruction.

This algorithm relies on various functions that can be used to configure its behavior. We need to define the function *merge_v*, which describes how to compound the state change of the current instruction and the current state, the function *merge_h*, which describes how to merge the states of several paths, the instruction analysis function *analyze_instr*. The function *pred*, which retrieves all possible predecessors of a block won't be implemented by us, because we rely on the DynInst instrumentation framework to achieve the following.

$$merge\_v : \mathcal{S}^{\mathcal{R}} \times \mathcal{S}^{\mathcal{R}} \mapsto \mathcal{S}^{\mathcal{L}} \tag{1a}$$

$$merge\_h : \mathcal{P}(\mathcal{S}^{\mathcal{R}}) \mapsto \mathcal{S}^{\mathcal{R}} \tag{1b}$$

$$analyze\_instr : \mathcal{I} \mapsto \mathcal{S}^{\mathcal{R}} \tag{1c}$$

$$pred : \mathcal{I} \mapsto \mathcal{P}(\mathcal{I}) \tag{1d}$$

As the *analyze_instr* function calculates the effect of an instruction and is the heart of the analyze function. It will also handle non jump and non fall-through successors, as these are not handled by DynInst in our case. We essentially have three cases that we handle: *1)* if the instruction is an indirect call or a direct call but we chose not to follow calls, then return a state where all trashed are considered written, *2)* if the instruction is a direct call and we chose to follow calls, then we spawn a new analysis and return its result, and *3)* in all other cases we simply return the decoded state.

This leaves us with the two merge functions remaining undefined and we will leave the implementation of these and the interpretation of the liveness state $\mathcal{S}^{\mathcal{L}}$ into parameters up to the following subsections.

Provided Parameter Count. To implement the *count* policy, we only need a coarse representation of the state of one register, thus we use the same representation as TypeArmor: *1)* T represents a trashed register, *2)* S represents a set register (written to), and *3)* U represents an untouched register.

This gives us the following register state $S^{\mathcal{L}} = \{T, S, U\}$ which translates to the register super state $\mathcal{S}^{\mathcal{R}} = (S^{\mathcal{R}})^{16}$.

We are only interested in the first occurrence of a S or T within one path, as following reads or writes do not give us more information. Therefore, our vertical merge function (*merge_v*) behaves in the following way that only when the first given state is U, is the return value the second state and in all other cases it will return the first state.

Our horizontal merge(*merge_h*) function is a simple pairwise combination of the given set of states, which are then combined with a union like operator with T preceding S preceding U.

The index of the highest parameter register based on the used call convention that has the state S is considered to be the number of parameters a callsite at most prepares.

Provided Parameter Wideness. To implement the *type* policy, we need a finer representation of the state of one register: *1)* T represents a trashed register, *2)* $s8, s16, s32, s64S$ represents a set register with 8-, 16-, 32-, 64-bit wideness, and *3)* U represents an untouched register.

This gives us the following register state $S^{\mathcal{L}} = \{T, s64, s32, s16, s8, U\}$ which translates to the register super state $\mathcal{S}^{\mathcal{R}} = (S^{\mathcal{R}})^{16}$.

Again, we are only interested in the first occurrence of a state that is not U in a path, as following reads or writes do not give us more information. Therefore, we can use the same vertical merge function as for the *count* policy, which is essentially a pass-through until the first non U state.

Our horizontal merge(*merge_h*) function is a simple pairwise combination of the given set of states, which are then combined with a union like operator with T preceding S preceding U. When both states are set, we pick the higher one.

Our experiments with this implementation showed two problems regarding provided wideness detection. Parameter lists with *holes* and address wideness underestimation, furthermore register extension instructions are also cause of problems. To reduce runtime, we also restricted the maximum path depth to 10 blocks.

Parameter Lists with Holes. This refers to parameter lists that show one or more `void` parameters between start to the last actual parameter. These are not existent in actual code but our analysis has the possibility of generating them through the merge operations. An example would be the following: A parameter list of $(64, 0, 64, 0, 0, 0)$ is concluded, although the actual parameter list might be $(64, 32, 64, 0, 0, 0)$. While the trailing 0es are what we expect, the 0 at the second parameter position will cause trouble, because it is an underestimation at the single parameter level, which we need to avoid. Our solution is to simply scan our reaching analysis result for these holes and replace them with the wideness 64, causing a (possible) overestimation.

Address Wideness Unterestimation. This refers to the issue that while in the callsite a constant value of 32-bit is written to a register, however the calltarget uses the whole 64-bit register. This can occur when pointers are passed from the callsite to the calltarget. Specifically this happens when pointers to memory inside the `.bss`, `.data` or `.rodata` section of the binary are passed. Our solution is to enhance our instruction analysis to watch out for constant writes. In case a 32-bit constant value write is detected, we check if the value is an address within the `.bss`, `.data` or `.rodata` section of the binary. If this is the case, we simply return a write access of 64-bit instead of 32-bit. This is not problematic, because we are looking for an overestimation of parameter wideness. It should be noted that the same problem can arise when a constant write causes the value 0 to be written to a 32-bit register. We use the same solution and set the wideness to 64-bit instead of 32-bit.

5 IMPLEMENTATION

We implemented TypeShield as a module pass for the *di-opt* environment pass provided by the DynInst [8] instrumentation framework (v.9.2.0). However, converting the pass to a standalone executable is also possible, as we do not rely on an extended set of DynInst features except for the pass abstraction. We currently restricted our analysis and instrumentation to x86-64 bit elf binaries using the SystemV call convention, because the DynInst library does not yet support the Windows platform. However, there is currently work going on in order to allow DynInst to work with Windows binaries as well. We focused on the SystemV call convention as most C/C++ compilers on Linux implement this ABI, however we encapsulated most ABI dependent behavior, so it should be possible to implement other ABIs with relative ease. Therefore, we deem it possible to implement TypeShield for the Windows platform in the near future, as we do not use any other platform-dependent API's. We developed the core part of our pass in an instruction analyzer, which relies on the DynamoRIO [1] library (v.6.6.1) to decode single instructions and provide access to its information. The analyzer is then used to implement our version of the reaching and liveness analysis (similar to PathArmor [26]), which can be customized with relative ease, as we allow for arbitrary path merging functions. However, we implemented the three basic versions as follows: destructive, intersection and union. To accomplish this we patched the DynInst library in order to allow for local annotation of calltargets with arbitrary information, leveraging its relocation schema, which relies on the basic block abstraction. We implemented a Clang/LLVM (v.4.0.0, trunk 283889) pass used for collecting ground truth data in order to measure the quality and performance of our tool. The ground truth data is then used to verify the output of our tool for several test targets. This is accomplished with the help of our python based evaluation and test environment. In total we implemented TypeShield in 5556 lines of code (LOC) of C++ code, our Clang/LLVM pass in 392 LOC of C++ code and our test environment in 3005 Python LOC.

6 EVALUATION

We evaluated TypeShield by instrumenting various open source applications and analyzing the results. We used the two ftp server applications *Vsftpd* (v.1.1.0) and *Proftpd* (v.1.3.3), the two http server applications *Postgresql* (v.9.0.10) and *Mysql* (v.5.1.65), the memory cache application *Memcached* (v.1.4.20) and the *Node.js* server application (v.0.12.5). We chose these applications, which are a subset of the applications also used by the TypeAmor [26] to allow for later comparison. In our evaluation we addressed the following research questions (RQs) w.r.t. TypeShield:

- **RQ1:** How **precise** is it? (§6.1)
- **RQ2:** How **effective** is it? (§6.2)
- **RQ3:** What is the **runtime overhead**? (§6.3)
- **RQ4:** What is the **instrumentation overhead**? (§6.4)
- **RQ5:** What **security level** does it offer? (§6.5)
- **RQ6:** Is it superior **compared** to other tools? (§6.6)

Comparison Method. As we do not have access (we requested the authors of TypeArmor several times to provide us access to the source code) to the source code of TypeAmor, we implemented two modes in TypeShield. The first mode of our tool is a similar implementation of the *count* policy described by TypeArmor. The second mode is our implementation of the *type* policy on top of our *count* policy implementation.

6.1 Precision

To measure the precision of TypeShield, we need to compare the classification of callsites and calltargets as is given by our tool to some sort of ground truth for our test targets. We generate this ground truth by compiling our test targets using a custom compiled Clang/LLVM compiler (v.4.0.0 trunk 283889) with a MachineFunction pass inside the x86 code generation implementation of LLVM. We essentially collect three data points for each callsite/calltarget from our LLVM-pass: *1)* the point of origination, which is either the name of the calltarget or the name of the function the callsite resides in, *2)* the return type that is either expected by the callsite or provided by the calltarget, and *3)* the parameter list that is provided by the callsite or expected by the calltarget, which discards the variadic argument list.

However, before we can proceed to measure the quality and precision of TypeShield's classification of calltargets and callsites using our ground truth, we need to evaluate the quality and applicability of the ground truth, we collected.

6.1.1 Quality and Applicability of Ground Truth. To assess the applicability of our collected ground truth, we essentially need to assess the structural compatibility of our two data sets. First, we take a look at the comparability of calltargets and second, we take a look at the compatibility of callsites. The results are depicted in Table 1.

O2	calltargets			callsites		
Target	match	Clang miss	tool miss	match	Clang miss	tool miss
ProFTPD	1189	13 (1.08%)	0 (0.0%)	148	0 (0.0)	0 (0.0)
VsFTPD	419	0 (0.0%)	0 (0.0%)	14	0 (0.0)	0 (0.0)
LightTPD	420	0 (0.0%)	0 (0.0%)	66	0 (0.0)	0 (0.0)
Nginx	1035	0 (0.0%)	0 (0.0%)	269	0 (0.0)	0 (0.0)
Postgres	7039	49 (0.69%)	0 (0.0%)	635	0 (0.0)	40 (0.0)
Memcached	248	0 (0.0%)	0 (0.0%)	48	0 (0.0)	0 (0.0)
geomean	850.33	1.97 (0.23%)	0.0 (0.0%)	101.92	0.0 (0.0)	0.85 (0.0)

Table 1: Table shows the quality of structural matching provided by our automated verify and test environment, regarding callsites and calltargets when compiling with optimization level O2. The label Clang miss denotes elements not found in the data-set of the Clang/LLVM pass. The label tool miss denotes elements not found in the data-set of TypeShield. TODO-add more description in order to indicate the advantage of our tool. What geomean values are good, low or hig? What is the main thing which can be observed if looking at this table?

Calltargets. The obvious choice for structural comparison regarding calltargets is their name, as these are simply functions. First, we have to remove internal functions from our data-sets like the `_init` or `_fini` functions, which are of no consequence for us. Furthermore, while C functions can simply be matched by their name as they are unique through the binary, the same can not be said about the language C++. One of the key differences between C and C++ is function overloading, which allows defining several functions with the same name, as long as they differ in namespace or parameter type. As LLVM does not know about either concept, the Clang compiler needs to generate unique names. The method used for unique name generation is called mangling and composes the actual name of the function, its return type, its name-space and the types of its parameter list. We therefore need to reverse this process and then compare the fully typed names. Table 1 shows three data points regarding calltargets for the optimization level O2: *1)* The number of comparable calltargets that are found in both data sets, *2)* Clang miss: The number of calltargets that are found by TypeShield but not by our Clang/LLVM pass, and *3)* tool miss: The number of calltargets that are found by our Clang/LLVM pass but not by TypeShield

The problematic column is the Clang miss column, as these might indicate problems with TypeShield. These numbers are relatively low (below 1%) with only Node.js showing a significant higher value than the rest (around 1.6%). The column labeled tool miss lists higher numbers, however these are of no real concern to us, as our ground truth pass possibly collects more data: All source files used during the compilation of our test-targets are incorporated into our ground truth. The compilation might generate more than one binary and therefore not necessary all source files are used for our test-target.

Considering this, we can safely state that our structural matching between ground truth and TypeShield regarding calltargets is nearly perfect (above 98%).

Callsites. While our structural matching of calltargets is rather simple, the matter of matching callsites is more complex. Our tool can provide accurate addressing of callsites within the binary. However, Clang/LLVM does not have such capabilities in its intermediate representation (IR). Furthermore the IR is not the final representation within the compiler, as the IR is transformed into a machine-based representation (MR), which is the again optimized. Although we can read information regarding parameters from the IR, it is not possible with the MR. Therefore, we attach that data directly after the conversion from IR to MR and read that data at the end of the compilation. To not unnecessarily pollute our data set, we only considered calltargets, which have been found in both data sets. Table 1 shows three data points regarding callsites for the optimization level O2: *1)* the number of comparable callsites that are found in both data sets, *2)* Clang miss: The number of callsites that are discarded from the data set of TypeShield, and *3)* tool miss: The number of callsites that are discarded from the data set of our Clang/LLVM pass.

Both columns (Clang miss and tool miss) show a relatively low number of problems (< 0.5%), therefore we can also safely state that our structural matching between ground truth and TypeShield regarding callsites is also nearly perfect (above 99%).

6.1.2 Classification Precision (count). We measured two data points per target, the number and ratio of perfect classifications and the number and ratio of problematic classifications, which in the case of calltargets refers to overestimations and in case of callsites refers to underestimations. The results are depicted in Table 2.

O2	Calltargets			Callsites		
Target	#	perfect	problem	#	perfect	problem
proftpd	1015	903 (88.96%)	0 (0.0%)	155	131 (84.51%)	0 (0.0%)
vsftpd	318	273 (85.84%)	0 (0.0%)	14	14 (100.0%)	0 (0.0%)
lighttpd	290	278 (95.86%)	0 (0.0%)	66	48 (72.72%)	0 (0.0%)
nginx	921	762 (82.73%)	0 (0.0%)	266	129 (48.49%)	0 (0.0%)
mysqld	9742	7195 (73.85%)	1 (0.01%)	7923	5138 (64.84%)	0 (0.0%)
postgres	6930	6433 (92.82%)	0 (0.0%)	687	536 (78.02%)	0 (0.0%)
memcached	133	123 (92.48%)	0 (0.0%)	48	40 (83.33%)	0 (0.0%)
node	20638	17427 (84.44%)	1 (0.0%)	10965	6288 (57.34%)	1 (0.0%)
geomean	1413.94	1228.29 (86.86%)	0.0 (0.0%)	319.7	230.12 (71.97%)	0.0 (0.0%)

Table 2: The results for analysis using the *count* policy on the O2 optimization level. TODO-add more description in order to indicate the advantage of our tool. What geomean values are good, low or hig? What is the main thing which can be observed if looking at this table?

Experiment Setup (Calltargets). Union combination operator with an *analyze* function that follows into occurring direct calls.
Results (Calltargets). The problem rate is under 0.01%, as there are only two test targets, that exhibit a problematic classification. The rate of perfect classification is in general over 80% with Mysql as an exception (73.85%) resulting in a geometric mean of 86.86%.
Experiment Setup (Callsites). Union combination operator with an *analyze* function that does not follow into occurring direct calls

while relying on a backward inter-procedural analysis. **Results (Callsites).** The problem rate is under 0.01%, as there is only one test target, that exhibit a problematic classification. The rate of perfect classification is in general over 60% with Nginx (48.49%) and Node.js (56.34%) as an exception resulting in a geometric mean of 71.97%.

6.1.3 Classification Precision (type). We measured two data points per test target, the number and ratio of perfect classifications and the number and ratio of problematic classifications, which in the case of calltargets refers to overestimations and in case of callsites refers to underestimations. The results are depicted in Table 3.

O2	Calltargets			Callsites		
Target	#	perfect	problem	#	perfect	problem
proftpd	1015	837 (82.46%)	10 (0.98%)	155	131 (84.51%)	0 (0.0%)
vsftpd	318	252 (79.24%)	3 (0.94%)	14	14 (100.0%)	0 (0.0%)
lighttpd	290	252 (86.89%)	1 (0.34%)	66	45 (68.18%)	1 (1.51%)
nginx	921	639 (69.38%)	0 (0.0%)	266	143 (53.75%)	8 (3.0%)
mysqld	9742	6154 (63.16%)	307 (3.15%)	7923	4391 (55.42%)	375 (4.73%)
postgres	6930	5691 (82.12%)	579 (8.35%)	687	476 (69.28%)	5 (0.72%)
memcached	133	109 (81.95%)	10 (7.51%)	48	43 (89.58%)	0 (0.0%)
node	20638	15483 (75.02%)	453 (2.19%)	10965	4909 (44.76%)	1038 (9.46%)
geomean	1413.94	1091.01 (77.15%)	22.0 (1.92%)	319.7	218.56 (68.35%)	7.97 (1.38%)

Table 3: The results for analysis using the *type* policy on the O2 optimization level. TODO-add more description in order to indicate the advantage of our tool. What geomean values are good, low or hig? What is the main thing which can be observed if looking at this table?

Experiment Setup (Calltargets). Union combination operator with an *analyze* function that does follow into occurring direct calls and a vertical merge that intersects all reads until the first write. **Results (Calltargets).** For half of the set, the problem rate is under 1% and for the other half it is not above 10%, resulting in a geomean of 1.92%. The rate of perfect classification is in general over 70% with Nginx (69.38%) and Mysql (63.16%) resulting in a geometric mean of 77.15%. **Experiment Setup (Callsites).** Union combination operator with an *analyze* function that does not follow into occurring direct calls while relying on a backward inter-procedural analysis. **Results (Callsites).** For two thirds of the set, the problem rate is under 2% and for last third it is not above 10%, resulting in a geomean of 1.38%. The rate of perfect classification is in general over 50% with Node.js (44.76%) as an exception resulting in a geometric mean of 68.35%.

6.2 Effectiveness

We are now going to evaluate the effectiveness of TypeShield leveraging the result of several experiment runs: First we are going to establish a baseline using the data collected from our Clang/LLVM pass, which are the theoretical limits our implementation can reach for both the *count* and the *type* schema. Second we are going to evaluate the effectiveness of our *count* policy and third we are going to evaluate the effectiveness of our *type* policy. For each series we collected three data points per test target, the average number of calltargets per callsite, the standard deviation σ and the median. The results are depicted in Table 4.

6.2.1 Theoretical Limits. We explore the theoretical limits regarding the effectiveness of the *count* and *type* policies by relying on the collected ground truth data, essentially assuming perfect classification. **Experiment Setup.** Based on the type information collected by our Clang/LLVM pass, we conducted two experiment series. We derived the available number of calltargets for each callsite based on the collected ground truth applying the *count* and *type* schema.

Results. *1)* The theoretical limit of the *count** schema has a geometric mean of 233 possible calltargets, which is 16.48% of the geometric mean of total available calltargets, and *2)* The theoretical limit of the *type** schema has a geometric mean of 210 possible calltargets, which is 14.86% of the geometric mean of total available calltargets.

When compared, the theoretical limit of the *type* policy allows about 10% less available calltargets in the geomean in O2 than the limit of the *count* policy.

6.2.2 Reduction achieved by TypeShield. **Experiment Setup.** We setup our two experiment series based on our previous evaluations regarding the classification precision for the *count* and the *type* policy.

Results. *1)* The *count* schema has a geometric mean of 315 possible calltargets, which is 22.29% of the geometric mean of total available calltargets. This is 35.19% more than the theoretical limit of available calltargets per callsite, and *2)* The *type* schema has a geometric mean of 290 possible calltargets, which is 20.52% of the geometric mean of total available calltargets. This is 38.09% more than the theoretical limit of available calltargets per callsite.

When compared, our implementation of the *type* policy allows about 7.93% less available calltargets in the geomean in O2 than our implementation of the *type* policy.

6.3 Runtime Overhead

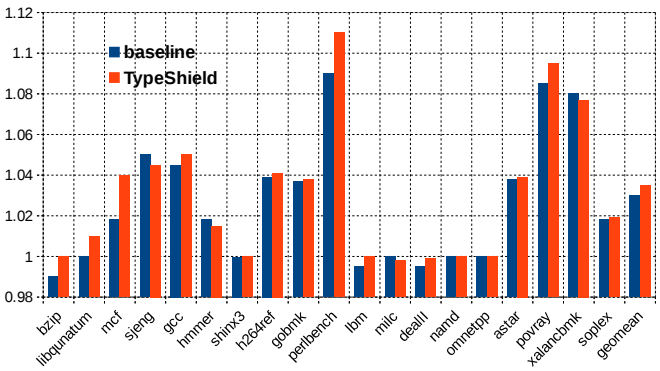

Figure 8: Benchmark run time normalized against the baseline for the SPEC CPU2006 benchmarks. TODO-add more description in order to indicate the advantage of our tool. What geomean values are good, low or hig? replace with our values. remove this stub.

Figure 8 depicts the runtime normalized against the baseline for the SPEC CPU2006 benchmarks. In general, we have usually about

O2 Target	AT	*count** limit (mean ± σ)	median	*count* limit (mean ± σ)	median	*type** limit (mean ± σ)	median	*type* limit (mean ± σ)	median
proftpd	390	349.31 ± 53.13	369.0	370.0 ± 43.59	382.0	333.12 ± 63.21	312.0	359.4 ± 54.0	348.0
vsftpd	10	7.14 ± 1.8	6.0	7.14 ± 1.8	6.0	5.42 ± 0.9	6.0	5.42 ± 0.9	6.0
lighttpd	59	34.87 ± 14.75	21.0	45.27 ± 14.31	59.0	32.33 ± 13.28	21.0	42.58 ± 14.58	59.0
nginx	543	318.62 ± 151.56	266.0	461.88 ± 128.12	543.0	318.62 ± 151.56	266.0	447.54 ± 132.37	543.0
mysqld	5883	4140.22 ± 1067.55	3167.0	4987.34 ± 948.74	5513.0	3899.92 ± 963.58	3167.0	4739.99 ± 933.25	5564.0
postgres	2491	2094.82 ± 634.24	2286.0	2194.84 ± 590.4	2340.0	1939.74 ± 771.02	2286.0	2060.44 ± 710.43	2332.0
memcached	14	12.31 ± 2.34	14.0	13.35 ± 1.1	14.0	10.29 ± 0.95	11.0	10.64 ± 1.05	10.0
node	7527	5119.4 ± 1548.08	5536.0	6430.54 ± 1279.63	5909.0	4394.4 ± 1516.75	3589.0	5788.81 ± 1444.1	4578.0
geomean	350.0	256.0 ± 76.0	233.0	298.0 ± 65.0	315.0	231.0 ± 69.0	210.0	270.0 ± 66.0	290.0

Table 4: The results of comparing our implementation results with the theoretical limits for the different restriction policies combined with an address taken analysis for optimization level O2. TODO-add more description in order to indicate the advantage of our tool. What geomean values are good, low or hig? What is the main thing which can be observed if looking at this table?

2%-5% performance drop when instrumenting using Dyninst. The reason for that are essentially cache misses introduced by jumping between the old and the new executable section of the binary generated by duplicating and patching the duplicate. This is necessary, because when out side of the compiler it is nigh on impossible to relocate indirect control flow, therefore every time an indirect control flow occurs, one jumps into the old executable section and from there back to the new executable section. Moreover, this is also dependent on the actual structure of the target, as it depends on the number of indirect control flow operations per time unit.

6.4 Instrumentation Overhead

The instrumentation overhead or the change in size due to patching is mostly due to the method Dyninst uses to patch binaries. Essentially the executable part of the binary is duplicated and extended with the patch. The usual ratio is around 40% to 60% while Postgres has an increase of 150% in binary size. One can not reduce that value significantly, because of the nature of code relocation after losing the data that a compiler has. Especially indirect control flow changes are very hard to relocate. Therefore, instead each important basic block in the old code contains a jump instruction to the new position of the basic block.

6.5 Security Analysis

Figures, 9, 10, 11, and 12 depict the CDFs for the following programs: Postgresql, Node.js, Proftpd, and Mysql when compiled with the -O2 Clang compiler flag. We selected these four programs randomly. The CDFs depict the number of legal callsite targets and the difference between the type and the count policies. While the count policies have only a few number of changes, the number of changes that can be seen within the type policies is vastly higher. The reason for that is simple, the number of buckets that are used to classify the callsites and calltargets is simply higher. While type policies mostly perform better than the count policies, there are still parts within the type plot that are above the count plot, the reason for that is relatively simple: the maximum number of calltargets a callsite can access has been reduced, therefore a lower amount of calltargets is a higher percentage than before. However, all these results are dependent on the structure of the program.

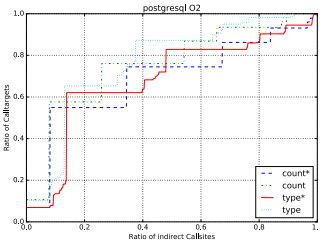

Figure 9: Postgresql -O2

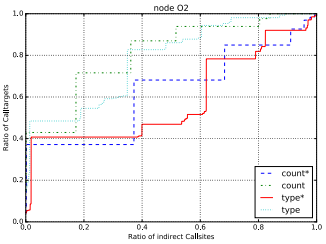

Figure 10: Node.js -O2

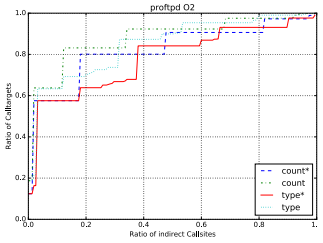

Figure 11: Proftpd -O2

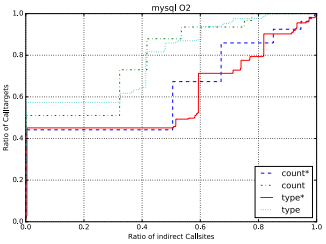

Figure 12: Mysql -O2

todo. Also, add the buckets diagram, see Figure 9 in the typearmor paper.

6.6 Comparison with Other Tools

Table 5 depicts a comparison between TypeShield, TypeArmor and IFCC w.r.t. the count of calltargets per callsites. The values depicted in this Table for TypeArmor and IFCC are taken from the original TypeArmor paper. We compare our version of address taken analysis (AT), TypeArmor, TypeShield (count), TypeShield (type) and IFCC. The first thing to notice is that when comparing these values, one can see that we did not implemented a separation based on return type or the CFC that TypeArmour introduced. Therefore, when implementing those measures, we predict that our solution would improve even more in w.r.t precision. While we think it is possible to surpass TypeArmor implementing those two solutions in our tool, we deem it nigh on impossible to be able

Target	AT	TypeArmor	IFCC	TypeShield (count)	TypeShield (type)
proftpd	390	376	3	382	348
vsftpd	10	12	1	6	6
lighttpd	59	47	6	59	59
nginx	543	254	25	543	543
mysqld	5883	3698	150	5513	5564
postgres	2491	2304	12	2340	2332
memcached	14	14	1	14	10
node	7527	4714	341	5909	4578
geomean	343.3	272.36	11.35	306.73	281.77

Table 5: The medians of calltargets per callsite for different programs. Note that smaller geomean values are better and that TypeShield currently does not include function return types as TypeArmor does. TODO-add more description in order to indicate the advantage of our tool. What geomean values are good, low or hig? What is the main thing which can be observed if looking at this table?

to compete with IFCC, which can directly operate on the source code level. Therefore, it has access to more possibilities than simply inspecting the parameters or return values.

7 LIMITATIONS

First, TypeShield is limited by the capabilities of the DynInst instrumentation environment, the main problem, we are facing here is that non returning functions like exit are not detected reliably in some cases, which is why we were not able to test the PureFTP server, as it heavily relies on these functions. The problem is that those non returning functions usually appear as a second branch within a function that occurs after the normal control flow, causing basic blocks from the following function to be attributed to the current function. This results in a malformed control flow graph and erroneous attribution of callsites and problematic miss classifications for both calltargets and callsites.

Second, TypeShield relies on variety within the binary, in particular we rely on the fact that functions use more than only 64-bit values or pointers within their parameter list. Should this scenario occur, our analysis has nothing to work with and essentially degrades into a parameter count based implementation. Thankfully this occurrence is quite rare, as we experienced within our experiments. When working based on source level information, we could not detect a difference between our *type* and a *count* policies. However, when leveraging our tool, we were able to detect differences, which reinforces the fact, that we do not rely on declaration of parameters but usage of those.

Third, TypeShield can protect only forward indirect edges in a binary program and is currently not intended to protect backward edges with the help of shadow stack [13]. For this reason we assume that TypeShield runs side by side with an an ideal backward-edge protection mechanism such as a shadow [11]. However, the main goal of TypeShield is to complement shadow stack based defenses which can not deal with attacks which do not violate the backward-edge calling conventions such as the COOP attack.

Fourth, TypeShield is not intended to be more precise than source code based tools such as IFCC/VTV [25]. On one hand, TypeShield is highly useful in situations where the source code for many off-the-shelf programs is not always available and where programs rely on many libraries and where the recompilation of all the shared libraries is not possible. On the other hand, binary based tools as TypeShield can offer precise protection when source code is not available or recompilation is not feasible or desirable.

Finally, TypeShield can not stop all possible attacks since even solutions with access to source code are unable to protect against all possible attacks [10]. Nevertheless, we show that TypeShield, our binary based tool can stop all COOP attacks published to date and significantly raises the bar for an adversary when compared to TypeArmor and other similar tools. Moreover, TypeShield provides a strong mitigation for other types of code-reuse attacks as well.

8 RELATED WORK

8.1 Advanced Code-Reuse Attacks Mitigation

Recursive-COOP [12], COOP [24], Subversive-C [19] and the attack of Lan *et al.* [18] are forward-edge based CRAs which can not be addressed with: *i)* with shadow stacks techniques (*i.e.,* do not violate the caller/calle convention), *ii)* coarse-grained Control-Flow Integrity (CFI) [6, 7] techniques are useless against these attacks, *iii)* hardware based approaches such as Intel CET [4] can not mitigate this attack for the same reason as in *i)*, and *iv)* with OS-based approaches such as Windows Control Flow Guard [3] since the precomputed CFG does not contain edges for indirect callsites which are explicitly exploited during the COOP attack. However, the following tools can protect against COOP attacks:

Binary based. vTable protection is addressed through binary instrumentation in tools such as: vfGuard [22], vTint [28]. However, none of these tools can help to mitigate against COOP. The only binary based tool which we are aware of that can mitigate protect against COOP is TypeArmor [26]. TypeArmor uses a fine-grained CFI policy based on caller (only indirect callsites)/callee matching which consists in checking during runtime if the number of provided and needed parameters match.

TypeShield is most similar to TypeArmor [26] since we also enforce strong binary-level invariants on the number of function parameters. TypeShield similarly to TypeArmor targets exclusive protection against advanced exploitation techniques which can bypass fine-grained CFI schemes and VTable protections at the binary level.

However, TypeShield offers a better restriction of calltargets to callsites, since we not only restrict based on the number of parameters but also on the wideness of their types. This results in much smaller buckets that in turn can only target a smaller subset of all address taken functions. However, we rely for that on the variety of parameter types and when there is none, we will degrade into a parameter count policy.

Source code based. Indirect callsite targets are checked based on vTable integrity. Different types of CFI policies are used such as in the following tools: SafeDispatch [15], IFCC/VTV [25] LLVM and GCC compiler. Additionally, the Redactor++ [12] uses randomization vTrust [27] checks calltarget function signatures, CPI [17] uses a memory safety technique in order to protect against the COOP attack.

There are several source code based tools which can successfully protect against the COOP attack. Such tools are: ShrinkWrap [14],

IFCC/VTV [25], SafeDispatch [15], vTrust [27], Readactor++ [12], CPI [17] and the tool presented by Bounov *et al.* [9]. These tools profit from high precision since they have access to the full semantic context of the program though the scope of the compiler on which they are based. Because of this reason, these tools target mostly other types of security problems than binary-based tools address. For example, some of the last advancements in compiler based protection against code reuse attacks address mainly performance issues. Currently, most of the above presented tools are only forward edge enforcers of fine-grained CFI policies with an overhead from 1% up to 15%.

We are aware that there is still a long research path to go until binary based techniques can recuperate program based semantic information from executable with the same precision as compiler based tools. This path could be even endless since compilers are optimized for speed and are designed to remove as much as possible semantic information from an executable in order to make the program run as fast as possible. In light of this fact, TypeShield is another attempt to recuperate just the needed semantic information (types and number of function parameters from indirect callsites) in order to be able to enforce a precise and with low overhead primitive against COOP attacks.

Rather than claiming that the invariants offered by TypeShield are sufficient to mitigate all versions of the COOP attack we take a more conservative path by claiming that TypeShield further raises the bar w.r.t. what is possible when defending against COOP attacks on the binary level.

9 CONCLUSION

In this paper, we presented TypeShield, a program binary based runtime fine-grained CFI enforcing tool which can mitigate forward indirect call based attacks by precisely filtering legitimate from illegitimate forward indirect control flow transfers in program binaries. TypeShield uses a novel runtime type checking technique based on function parameter type checking and parameter counting in order to efficiently filter-out legitimate and illegitimate forward indirect transfers. It provides a more precise analysis then existing approaches with a comparable performance overhead. We have implemented it and applied it to real software such as web servers and FTP servers. We demonstrated through extensive experiments and comparisons with related tools that TypeShield has higher precision and comparable performance overhead than existing state-of-the-art tools. To date, we were able to provide a more precise technique than parameter count based techniques by reducing the possible calltargets per callsite ratio by 20% with an overall reduction of about 9% when comparing with similar state-of-the-art approaches. The outcome is a more precise analysis and a considerably reduced attack surface. In the spirit of open research, we have made the source code of TypeShield publicly available at https://github.com/stub/typeshield.

REFERENCES

[1] DynamoRIO. http://dynamorio.org/home.html.
[2] 2015. BlueLotus Team, Bctf challenge: bypass vtable read-only checks.
[3] 2015. Windows Control Flow Guard. http://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx
[4] 2016. Intel Control-flow Enforcement Technology. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.
[5] 2016. Working Draft, Standard for ProgrammingLanguage C++ N4618. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4618.pdf.
[6] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*.
[7] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control Flow Integrity Principles, Implementations, and Applications. In *ACM Transactions on Information and System Security (TISSEC)*.
[8] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, (PASTE)*.
[9] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Symposium on Network and Distributed System Security (NDSS)*.
[10] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the USENIX conference on Security (USENIX SEC)*.
[11] Mauro Conti, Per Larsen, Stephen Crane, Lucas Davi, Michael Franz, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[12] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[13] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.
[14] Istvan Haller, Enes Goktas, Elias Athanasopoulos, G. Portokalidis, and Herbert Bos. 2015. ShrinkWrap: VTable Protection Without Loose Ends. In *Annual Computer Security Applications Conference (ACSAC)*.
[15] D. Jang, T. Tatlock, and S. Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Symposium on Network and Distributed System Security (NDSS)*.
[16] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2009. *Data flow analysis: Theory and Practice*. CRC Press.
[17] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[18] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. 2015. Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses. In *IEEE Trustcom/BigDataSE/ISPA*.
[19] Julian Lettner, Benjamin Kollenda, Andrei Homescu, Per Larsen, Felix Schuster, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Michael Franz. 2016. Subversive-C: Abusing and Protecting Dynamic Message Dispatch. In *USENIX Annual Technical Conference (USENIX ATC)*.
[20] US NIST National Vulnerability Database (NVD). 2017. In total 701 arbitrary code executions reported, Jan.'08 to May'17. https://nvd.nist.gov/vuln/search/results?adv_search=true&form_type=advanced&results_type=overview&query=arbitrary+code+execution&pub_date_start_month=0&pub_date_start_year=2008&pub_date_end_month=6&pub_date_end_year=2017.
[21] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-Grained Control-Flow Integrity through Binary Hardening. In *DIMVA*.
[22] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Symposium on Network and Dist—ributed System Security (NDSS)*.
[23] G. Ramalingam. 1994. The Undecidability of Aliasing. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*.
[24] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-Oriented Programming. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.
[25] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM. In *Proceedings of the USENIX conference on Security (USENIX SEC)*.
[26] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P),*.
[27] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. 2016. VTrust: Regaining Trust on Virtual Calls. In *Symposium on Network and Distributed System Security (NDSS)*.

[28] Chao Zhang, Chengyu Song, Kevin Chen Zhijie, Zhaofeng Chen, and Dawn Song. 2015. vTint: Protecting Virtual Function TablesÍntegrity. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[29] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[30] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the USENIX conference on Security (USENIX SEC)*.