TypeShield: Practical Forward-Backward Edge Attack 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 support 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 control-flow integrity (CFI) policy 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/callee matching (*i.e.,* precision), while maintaining a more favorable runtime performance overhead ($\leq 2\%$) than other state-of-the-art tools. 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 35% associated with an overall reduction of about 13% and a lower runtime performance overhead as other state-of-the-art parameter-only count-based tools. Finally, we want to particularly emphasize that in this paper we provide for each experiment a precise description w.r.t. setup, results, tool misses, mean, median and geomean values which clearly increases the reproducibility.

KEYWORDS

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

1 INTRODUCTION

The object-oriented programming (OOP) paradigm and the C++ programming language are the de facto standard for developing large, complex and efficient software systems, in particular, when runtime performance and reliability are primary objectives.

A key building block of (runtime) OOP polymorphism are virtual functions, which enable late binding and allow programmers to overwrite a virtual function of the base-class with their own implementations. In order to implement virtual functions, the compiler needs to generate a virtual table meta-data structure of all virtual functions for each class containing them and provide to each instance of such a class a (virtual) pointer to the aforementioned table. While this approach allows for more flexible code to be built, the basic implementation provides unfortunately very little security assurances. Data about highly damaging arbitrary code executions in major applications collected by U.S. NIST (see Figure 1 [1] and [22]) demonstrates the security shortcomings and the need to address this problem space.

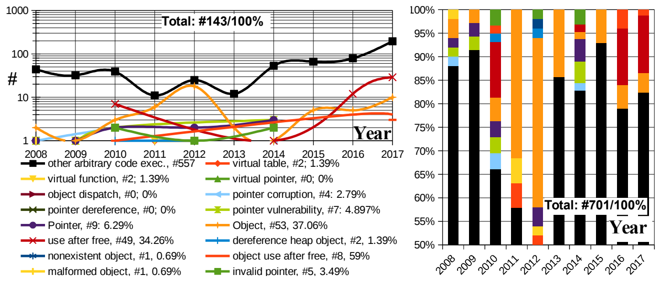


Figure 1: 10 Years of Arbitrary Code Reuse Attacks Statistics.

While the reasons for unwanted outcomes can be highly diverse, our work is primarily motivated by memory corruption attacks (*e.g.,* buffer/integer overflows), which can enable the execution of sophisticated Code-Reuse Attacks (CRAs) such as the advanced COOP attack [25] and its extensions [2, 10, 19, 21]. A necessary ingredient for this class of attacks is the ability to corrupt the virtual pointer of an object.

To address such object dispatch corruptions, Control-Flow Integrity (CFI) [5, 6] was originally developed to secure indirect control flow transfers by adding runtime checks before each indirect callsite. Unfortunately, COOP and its brethren bypass most deployed CFI-based enforcement policies, since these attacks do not exploit indirect backward edges (*i.e.,* return edges), but rather exploit the forward indirect control flow transfer imprecision which cannot be determined statically upfront as alias analysis is undecidable [24] in program binaries.

More recent techniques and tools can be distinguished into those relying on *source code* access including SafeDispatch [15], ShrinkWrap [14], VTI [7], and IFCC/VTV [26]; the latter being used in production, but the reliance on source-code availability limits the applicability of the approach. In contrast, *binary*-based tools typically rely on forward-edge CFI policies. Examples include binCFI [30, 31], VCI [12] and TypeArmor [27].

According to our assessment of the literature, TypeArmor serves as the state-of-the-art of binary-based defense tools against advanced CRAs. In TypeArmor, a fine-grained forward edge CFI-policy based on parameter count for binaries is implemented. It calculates invariants for calltargets and indirect callsites based on

[1] Number (#, left side of Figure 1) and percentage (%, right side of Figure 1) 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 Figure 1)* reported by U.S. NVD for the past 10 years [22]. 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 result of a vptr/vtable corruption (see * above) that are exploited by highjacking forward indirect calls. These vulnerabilities were reported in applications such as Google's Chrome & V8 JavaScript engine, Mozilla Firefox, Microsoft's IE 10, Edge & Chakra JavaScript engine, and several iOS/MacOS applications.

the number of parameters they use by leveraging static analysis of the binary, which then is patched to enforce those invariants during runtime. While we believe the general approach to be highly promising, we consider as a significant shortcoming that TypeArmor lacks precision with respect to the number of calltargets allowed per callsite which introduces significant inefficiencies (see §11.3 for more details). With our work, we aim to achieve both significant precision and runtime efficiency enhancements.

In this paper, we present TypeShield, a runtime binary-level fine-grained CFI tool for illegitimate forward calls, that is based on an improved forward-edge fine-grained CFI-policy compared to previous work [27]. TypeShield takes the binary of a program as input and it automatically instruments it in order to detect illegitimate indirect calls at runtime. In order to achieve this, TypeShield analyzes 64-bit binaries by focusing on function parameters which are passed with the help of registers. Based on the used ABI, TypeShield is consequently 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. As we demonstrate in the evaluation section, this setup provides us with enough information to be significantly more precise than [27] when aiming to stop several state-of-the-art CRAs.

Analysis Description. 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 [27] for all calltargets, and *3)* compares individually parameters of callsites and calltargets in order to check if an indirect calltransfer is acceptable or not, thereby providing a more fine-grained calltarget set per callsite compared to other state-of-the-art tools. TypeShield uses automatically inferred parameter types which are used to construct a more precise construction of both the callee parameter types and callsite signatures. This is later used in the classification of matching callsites and calltargets. The result is a more precise callee target set for each caller than TypeArmor.

Analysis Details. The TypeShield analysis is based on a use-def callees analysis to approximate the function prototypes, and liveness analysis at indirect callsites to approximate callsite signatures. This efficiently leads to a more precise control flow graph (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.

Used Policy. 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 callsite are not used by any callee by design. This further helps to reduce the possible target set of callees for each callsite.

Comparison. TypeShield, compared to TypeArmor, uses different analysis strategies for basic block merging. Furthermore, TypeShield (disallows an indirect calltransfer 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 callsite targets only a strict calltarget set. Finally, the program binary hardened by TypeShield contains a considerably reduced available calltarget set per callsite, thus drastically limiting an attacker in his capabilities.

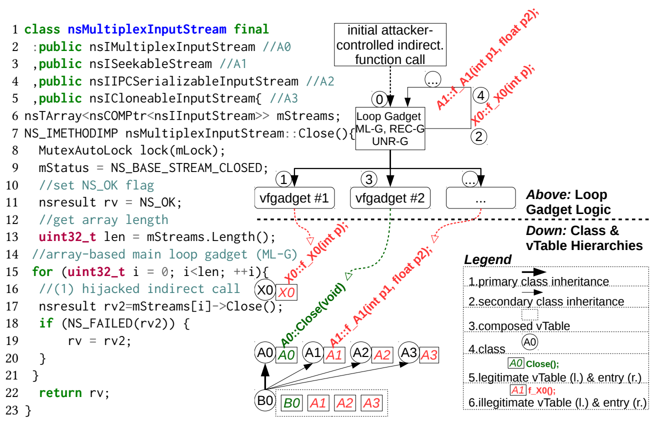
In summary, we make the following contributions:

- We provide a thorough **security analysis of forward indirect calls**. We analyze the usage of illegitimate indirect forward calls in detail, thus providing security researchers and practitioners a better understanding of this emerging threat (see 11.1, 11.2, 11.3, and 11.4 for more details).
- We designed and implemented an **illegitimate indirect calls detection tool**. TypeShield is 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. Further, TypeShield has an expanded **scope** of analysis. Our tool can detect forbidden indirect calls and as such it can protect, similarly as vTrust [28], against virtual table injection, corruption and reuse attacks. Further it can protect also against the control flow jujutsu attack [13] since our policy checks for void and non-void functions. As such TypeShield can serve as a platform for developing other types of defenses for different types of attacks. The source code, test scripts and evaluation results are available at https://github.com/domain/typeshield.
- We conduct a thorough set of evaluative **experiments**. In particular, we demonstrate experimentally that our precise binary-level CFI technique can mitigate advanced code reuse attacks (CRAs) in absence of C++ semantics. For example, TypeShield can effectively protect against the COOP attack and its variations. Thereby, it achieves a high degree of **precision**. Specifically, it employs a more precise analysis than TypeArmor in order to reduce the calltarget set for each callsite. Our evaluation shows that it gains between 13% and 35% more precision with respect to TypeArmor on the same programs. Further, we showcase similar **performance** enhancements in comparison to prior work. TypeShield employs runtime policy optimization techniques to further reduce runtime overheads. This is necessary, since our runtime checks are more complex than those used by TypeArmor. Our evaluation shows that it imposes up to 2% overheads for performance-intensive benchmarks on the SPEC CPU2006 benchmarks and web server applications, respectively. Which is lower than TypeArmors runtime overhead on the same programs. Finally, we respond to calls emphasizing the importance of reproducibility of evaluation results (see NISTIR 7564 [16]) by providing for each conducted experiment a precise description of encountered problems, setup, and comprehensive results including mean, median, and geomean values.

2 FORBIDDEN FORWARD CALLS EXPOSED

2.1 Exploiting Object Dispatches

Figure 2 depicts a C++ code example where it is illustrated how a COOP loop based gadget (*i.e.,* based on ML-G, REC-G, UNR-G, see [10]) works. Each `vfgadget` ① can be called in several ways, see ML-G, REC-G and UNR-G in Figure 2. 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 of a different virtual table (vTable) entry contained in the: *1)* class hierarchy (overall, whole program), *2)* class hierarchy (partial, only



```
class nsMultiplexInputStream final
 :public nsIMultiplexInputStream //A0
 ,public nsISeekableStream //A1
 ,public nsIIPCSerializableInputStream //A2
 ,public nsICloneableInputStream{ //A3
nsTArray<nsCOMPtr<nsIInputStream>> mStreams;
NS_IMETHODIMP nsMultiplexInputStream::Close(){
  MutexAutoLock lock(mLock);
  mStatus = NS_BASE_STREAM_CLOSED;
  //set NS_OK flag
  nsresult rv = NS_OK;
  //get array length
  uint32_t len = mStreams.Length();
 //array-based main loop gadget (ML-G)
 for (uint32_t i = 0; i<len; ++i){
  //(1) hijacked indirect call
  nsresult rv2=mStreams[i]->Close();
  if (NS_FAILED(rv2)) {
      rv = rv2;
  }
 }
  return rv;
}
```

initial attacker-controlled indirect. function call
0
...
4
2
A1::f_A1(int p1, float p2);
X0::f_X0(int p);
Loop Gadget ML-G, REC-G UNR-G
1
3
...
vfgadget #1
vfgadget #2
...
Above: Loop Gadget Logic
Down: Class & vTable Hierarchies
X0::f_X0(int p);
A0::Close(void)
A1::f_A1(int p1, float p2);
X0
A0
A1
A2
A3
B0
Legend
1.primary class inheritance
2.secondary class inheritance
3.composed vTable
4.class
A0 Close();
5.legitimate vTable (l.) & entry (r.)
A1 f_X0();
6.illegitimate vTable (l.) & entry (r.)

Figure 2: A COOP main loop gadget (ML-G) at work.

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 are 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 any possible entry 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

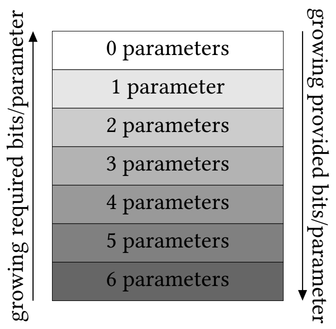


Figure 3: Call(sites/targets) *count* policy class. schema.

We build our own *count* policy which essentially resembles the policy introduced by TypeArmor [27] and for this reason we opted to briefly present it herein. 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 do not consider 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 which we will explore within this work.

3 OVERVIEW

3.1 Adversary Model and Assumptions

We largely use the same threat model and the same basic assumptions as described in [27]. More precisely, we assume a resourceful attacker that has read and write access to the data sections of the attacked program 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. Further, we assume that our attacker has the ability to execute a memory corruption to hijack the program control flow. Finally, the analyzed program binary is not hand-crafted and the compiler which was used to generate the binary adheres to one of the standard calling conventions mentioned in the first section of this paper.

3.2 Invariants for Calltargets and Callsites

Advanced code reuse attacks use different calltargets than the ones expected by an indirect callsite. As standard CFI solutions can hardly restrict these calltarget sets, TypeArmor proposes two base invariants: *1)* indirect callsites provide a number of parameters (*i.e.,* possibly overestimated compared to program source code), and *2)* calltargets require a minimum number of parameters (*i.e.,* possibly underestimated compared to program source code). The basic idea is that a callsite might only call functions that do not require more parameters than provided by the callsite. Finally, 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

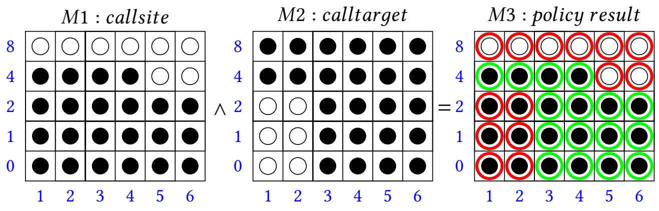


Figure 4: TypeShield's parameter type and count policy. The X and Y axis of matrices $M1, M2$ and $M3$ represent function parameter count and bit-widths in bytes, respectively. Note that our type policy performs an $\wedge$ (*i.e.,* logical and) operation between each entry in $M1_{i,j}$ and $M2_{i,j}$ 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 $pcs1,...,pcs6$ 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 $pct1,...,pct6$ 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 calltarget ct_A and B a callsite 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 calltarget ct_A and B a callsite cs_B than: $ct_A \subseteq cs_B \iff \forall i \subseteq [1,6]$, count(parameter($A$)) $\leq$ count(parameter(B)). From Definitions (3.3) and (3.3) 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.

3.4 Impact of TypeShield on COOP

Figure 5 represents a sub-part of the total indirect transfers space in any given C/C++ program represented as a lattice. 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 call-target as long as the number of parameters provided and required are fulfilling 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. Also note that the number of provided parameters must be not lower than the requirement 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.

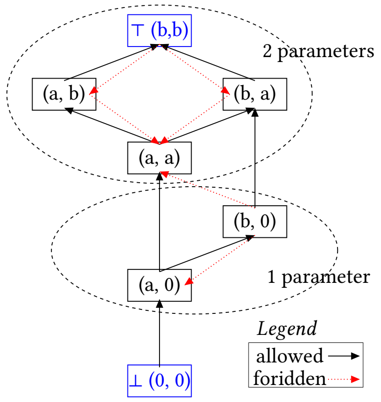


Figure 5: Transition lattice between call(targets/callsites). $a \wedge b \in \{0-bit, 8-bit, 16-bit, 32-bit, 64-bit\}$ and the two function parameters (for brevity) having $\{0-byte, 1-byte, 2-byte, 4-byte, 8-byte\}$ register wideness. TypeShield allows a transition from $a \rightarrow b \; iff \; a_i \leq b_i$ 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. TypeShield allows only the legal transfers, in contrast [27] allows all.

4 DESIGN

In this section, we cover the design of TypeShield. First, we present the theory and definitions for our instructions analysis based on register states in §4.1. Second, we present the details of our *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 §4.5.

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 $i \in$ `INSTR` can non-exclusively perform two kinds of operations on any number of existing registers:[2] *1)* Read n-bit from the register with $n \in \{64, 32, 16, 8\}$, and *2)* Write n-bit to the register with

[2]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.

$n \in \{64, 32, 16, 8\}$. We describe the possible change within one register as $\delta \in \Delta$ with $\Delta = \{w64, w32, w16, w8, 0\} \times \{r64, r32, r16, r8, 0\}$. [3] 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 $i \in$ INSTR via the function *decode* : INSTR $\mapsto \Delta^{16}$.

4.2 *Type* Policy

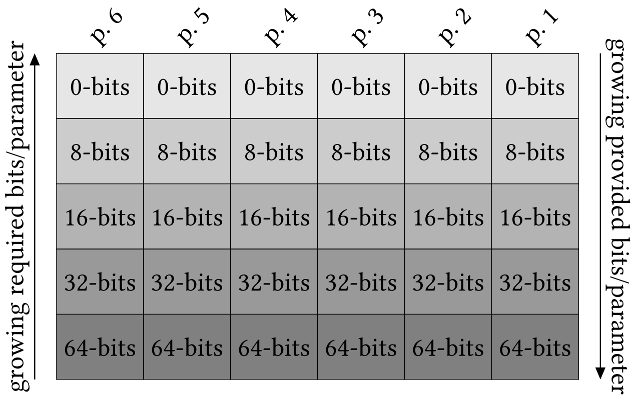

p. 6
p. 5
p. 4
p. 3
p. 2
p. 1
0-bits
8-bits
16-bits
32-bits
64-bits
growing required bits/parameter
growing provided bits/parameter

Figure 6: The *type* policy schema for callsites and calltargets. Note that p. means parameter. As it is depicted in this example, when requiring parameter width, one starts at the bottom of the above matrix and grows to the top, as it is always possible to accept more parameters than required. The reverse is true for providing parameters, as it is possible to accept less parameters 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 also on the parameter type. To simplify our approach, we use the width of the type and do not infer the actual type. As previously mentioned, there are 4 types of reading and writing accesses. Therefore, our set of possible types for parameters is TYPE $= \{64, 32, 16, 8, 0\}$; where 0 models the absence of a parameter. Since SystemV ABI specifies 6 registers as parameter holding registers, we classify our callsites and calltargets into TYPE^6. Similar to the policy of TypeArmor, we allow overestimations of callsites and underestimations of calltargets, however, on the level of types. Therefore, for a callsite cs it is possible to call a calltarget ct, only if for each parameter of ct the corresponding parameter of cs is not smaller w.r.t. the width. This results in a finer-grained policy which is further restricting the possible pool of calltargets for each callsite.

4.3 Calltarget Analysis

For our policy, we need to classify 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.

4.3.1 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 the variables that need to be alive at the beginning, as these are its parameters. We based Algorithm 1 on the liveness analysis algorithm presented in Khedker *et al.* [18], which basically is a depth-first traversal of basic 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 of:

- `merge_v` : $\mathcal{S}^{\mathcal{L}} \times \mathcal{S}^{\mathcal{L}} \mapsto \mathcal{S}^{\mathcal{L}}$, which describes how to merge the current state with the following state change.
- `merge_h` : $\mathcal{P}(\mathcal{S}^{\mathcal{L}}) \mapsto \mathcal{S}^{\mathcal{L}}$, which describes how to merge a set of states resulting from several paths.
- `analyze_instr` : INSTR $\mapsto \mathcal{S}^{\mathcal{L}}$, which calculates the state change that occurs due to the given instruction
- `succ` : INSTR* $\mapsto \mathcal{P}$(INSTR*), which calculates the successors of the given block.

Algorithm 1: Basic block liveness analysis.

Input : The basic block to be analyzed - block : INSTR*
Output : The liveness state - $\mathcal{S}^{\mathcal{L}}$

```
Function analyze (block : INSTR*) : S^L is
    state = Bl ;                          ▷ Initialize the state
    foreach inst ∈ block do
        state' = analyze_instr(inst) ; ▷ Calculate changes
        state = merge_h(state, state') ;      ▷ Merge changes
    end
    states = {} ;                  ▷ Set of surccessor states
    blocks = succ(block) ;         ▷ Get successors(succ.)
    foreach block' ∈ blocks do
        state' = analyze(block') ;  ▷ Analyze succ. block
        states = states ∪ { state' } ;     ▷ Add succ. states
    end
    state' = merge_h (states) ;          ▷ Merge succ. states
    return merge_v(state, state') ; ▷ Merge to final state
end
```

In our specific case, the function `analyze_instr` needs to also handle non-jump and non-fall-through successors, as these are not handled by DynInst. Essentially, there are three 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. 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. 3) In any other case, we simply return the decoded state. This leaves us with the two undefined merge

[3]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.

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

4.3.2 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 superstate is only updated if the corresponding register has yet to be accessed, as represented by C. Our reasoning is that the first access is the relevant one to determine read before write.

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 W preceding R preceding C.

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.

4.3.3 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 width, and *3)* C represents the absence of access. This gives us the following $S^{\mathcal{L}} = \{C, r8, r16, r32, r64, W\}$ register state 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 permit 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 with the functionality of an end marker. 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.

4.3.4 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 $stdout$. This type of functions allows for an easier processing of parameters where usually all potential variadic parameters are moved into a contiguous block of memory, as can be observed in the assembly depicted in Figure 10 in Appendix. Our analysis interprets that as a read access on all parameters and thus, we arrive at a potentially problematic overestimation.

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 our desired outcome using the following steps: *1)* we search for (what we call) the xmm-passthrough block, which entirely consists of moving 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.

4.3.5 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 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.

4.4 Callsite Analysis

Four our policy, we need to classify our callsites according to the parameters they provide. Overestimations are allowed, however, underestimations shall not be permitted. For this purpose we empoly a customizable modified reaching definition algorithm, which we will show first. We then presen our version for a *count* and a *type* policy. Furthermore, we need to be awayre of certain corner cases, which we will discuss at the end.

4.4.1 Reaching Definitions. An assignment to a variable is a reaching definition after the execution of a set of instruction if that variable still exists in at least one possible execution path. If applied to a callsite, this calculates the values that are provided by this callsite to the function it then invokes. We based Algorithm 2 on the reaching defintion analysis presented in Khedker *et al.* [18], which basically is a reverse depth-first traversal of basic blocks of a program. For customization, we rely on the implementation of several functions. $\mathcal{S}^{\mathcal{R}}$ is the set of possible register states depending on the specific reaching definition implementation of:

• `merge_v` : $\mathcal{S}^{\mathcal{R}} \times \mathcal{S}^{\mathcal{R}} \mapsto \mathcal{S}^{\mathcal{R}}$, which describes how to merge the current state with teh following state change.

• `merge_h` : $\mathcal{P}(\mathcal{S}^{\mathcal{R}}) \mapsto \mathcal{S}^{\mathcal{R}}$, which describes how to merge a set of states resulting from several paths.

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

• `pred` : `INSTR`* $\mapsto \mathcal{P}($`INSTR`$^*)$, which calculates the predecessors of the given block.

4.5 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

Algorithm 2: Basic block reaching definition analysis.

```
Input  : The basic block to be analyzed - block : INSTR*
Output : The reaching definition state - $\mathcal{S}^{\mathcal{R}}$
Function analyze(block : INSTR*) : $\mathcal{S}^{\mathcal{R}}$ is
    state = Bl ;                                ▷ Initialize the state
    foreach inst ∈ reversed(block) do
        state' = analyze_instr(inst) ;          ▷ Calculate changes
        state = merge_v(state, state') ;        ▷ Merge changes
    end
    states = {} ;                               ▷ Set of predecessor states
    blocks = pred(block) ;                      ▷ Get predecessors(pred.)
    foreach block' ∈ blocks do
        state' = analyze(block') ;              ▷ Analyze pred. block
        states = states ∪ { state' } ;          ▷ Add pred. states
    end
    state' = merge_h (states) ;                 ▷ Merge pred. states
    return merge_v(state, state') ;             ▷ Merge to final state
end
```

customizable, we look at the required merge functions to implement the *count* and *type* policies.

4.5.1 Reaching Definitions. 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.

In our specific case, the function `analyze_instr` does not need to handle normal predecessors, as Dyninst will resolve those for us. However there are serveral instructions that have to be handled nonetheless: 1) If the current instruction is an indirect call or a direct call but and the chosen implementation should not follow calls, then return a state where all registers are considered trashed. 2) If the instruction is a direct call and the chosen implementation should follow calls, then we start an analysis of the target function. 3) In all other cases we simply return the decoded state. This leaves us with the two merge functions and the undefined reaching definitions state $\mathcal{S}^{\mathcal{R}}$. In the following two paragraphs we will present two implementation variants: first similar to TypeArmour a *count* based policy and second our *type* based policy.

Previous work [18] provides a reaching definition analysis on blocks, which we use to arrive at the 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*. Note, that the function *pred*, which retrieves all possible predecessors of a block has not been implemented by us, because we rely on the DynInst instrumentation framework to achieve the following.

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

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

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

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

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.

4.5.2 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 $S^{\mathcal{L}} = \{T, S, U\}$ register state 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: 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.

4.5.3 Provided Parameter Width. In order 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 width, and *3)* U represents an untouched register. This gives us the following $S^{\mathcal{L}} = \{T, s64, s32, s16, s8, U\}$ register state 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 the provided width detection. Parameter lists with *holes* and address width 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 0s 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 scan our reaching analysis result for these holes and replace them with the wideness 64, causing a (possible) overestimation.

Address Width Unterestimation. This refers to the issue that while in the callsite a constant value of 32-bit is written to a register, 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 width to 64-bit instead of 32-bit.

5 IMPLEMENTATION

We implemented TypeShield using the instrumentation framework (v.9.2.0). 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 ongoing work 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 APIs. 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 [27]), which can be customized with relative ease, as we allow for arbitrary path merging functions. 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 5501 lines of code (LOC) of C++ code, our Clang/LLVM pass in 416 LOC of C++ code and our test environment in 3239 Python LOC.

6 EVALUATION

We evaluated TypeShield by instrumenting various open source applications and conducting a thorough analysis. Our test sample includes 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 in the TypeAmor paper [27], to allow for comparison. We addressed the following research questions (RQs).

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

Comparison Method. As we do not have access to the source code of TypeArmor (Note: we asked the authors of TypeArmor several times to provide us access to the source code), 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 the *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 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.*

We assessed the applicability of our collected ground truth, by assessing the structural compatibility of our two data sets. First, we investigate the comparability of calltargets. Second, we check 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	1202	0 (0.0%)	1 (0.08%)	157	0 (0.0)	0 (0.08)
pure-ftpd	276	1 (0.36%)	0 (0.0%)	8	2 (20.0)	5 (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)
mysqld	9952	9 (0.09%)	7 (0.07%)	8002	477 (5.62)	52 (0.07)
postgres	7079	9 (0.12%)	0 (0.0%)	635	80 (11.18)	40 (0.0)
memcached	248	0 (0.0%)	0 (0.0%)	48	0 (0.0)	0 (0.0)
node	20337	926 (4.35%)	23 (0.11%)	10502	584 (5.26)	261 (0.11)
geomean	1405.98	2.84 (0.27%)	0.93 (0.02%)	210.15	6.41 (1.8)	4.32 (0.02)

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.

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 datasets like the `_init` or `_fini` functions, which are of no relevance for this investigation. Furthermore, while C functions can simply be matched by their name as they are unique through the binary, the same cannot 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. Therefore, we 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)* Number of comparable calltargets that are found in both datasets, *2)* Clang miss: Number of calltargets that are found by TypeShield, but not by our Clang/LLVM pass, and *3)* Tool miss: 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 values might indicate problems with TypeShield. These numbers are relatively low (below 1%) with only Node.js showing a noticeable 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 state with confidence 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 again optimized. Although, we can read information regarding parameters from the IR, it is not possible with the MR. Therefore, we extract 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 dataset, we only considered calltargets, which have been found in both datasets. Table 1 shows three data points regarding callsites for the optimization level O2: *1)* Number of comparable callsites that are found in both datasets, *2)* Clang miss: Number of callsites that are discarded from the data set of TypeShield, and *3)* Tool miss: 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 close to perfection (above 99%).

6.1.2 *Classification Precision (count).*

We measured two data points per target, the number and ratio of perfect classifications

O2	Calltargets			Callsites		
Target	#	perfect args	perfect return	#	perfect args	perfect return
proftpd	1009	898 (88.99%)	845 (83.74%)	157	130 (82.8%)	113 (71.97%)
pure-ftpd	128	107 (83.59%)	52 (40.62%)	8	4 (50.0%)	8 (100.0%)
vsftpd	315	270 (85.71%)	193 (61.26%)	14	14 (100.0%)	14 (100.0%)
lighttpd	289	277 (95.84%)	258 (89.27%)	66	48 (72.72%)	57 (86.36%)
nginx	913	753 (82.47%)	777 (85.1%)	269	150 (55.76%)	232 (86.24%)
mysqld	9728	7138 (73.37%)	7845 (80.64%)	8002	5244 (65.53%)	6449 (80.59%)
postgres	6873	6378 (92.79%)	5241 (76.25%)	635	500 (78.74%)	573 (90.23%)
memcached	133	123 (92.48%)	77 (57.89%)	48	47 (97.91%)	48 (100.0%)
node	20069	16853 (83.97%)	14652 (73.0%)	10502	6001 (57.14%)	8841 (84.18%)
geomean	1074.9	928.04 (86.33%)	754.09 (70.14%)	210.15	150.13 (71.43%)	185.68 (88.35%)

Table 2: The results for classification of callsites and calltargets using our *count* policy on the O2 optimization level, when comparing to the ground truth obtained by our Clang/LLVM pass. The label perfect args denotes all occurrences when our result and the ground truth perfectly match regarding the required/provided arguments. The label perfect return denotes this for return values.

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 (see Table 2 for more details). **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 exhibits a problematic classification. The rate of perfect classification is in general over 60% with Nginx (48.49%) and Node.js (56.34%) as exceptions 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. **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 the 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%.

O2	**Calltargets**			**Callsites**		
Target	#	perfect args	perfect return	#	perfect args	perfect return
proftpd	1009	835 (82.75%)	861 (85.33%)	157	125 (79.61%)	113 (71.97%)
pure-ftpd	128	101 (78.9%)	54 (42.18%)	8	4 (50.0%)	8 (100.0%)
vsftpd	315	256 (81.26%)	179 (56.82%)	14	14 (100.0%)	14 (100.0%)
lighttpd	289	253 (87.54%)	244 (84.42%)	66	48 (72.72%)	57 (86.36%)
nginx	913	639 (69.98%)	753 (82.47%)	269	140 (52.04%)	232 (86.24%)
mysqld	9728	6141 (63.12%)	7684 (78.98%)	8002	4477 (55.94%)	6449 (80.59%)
postgres	6873	5730 (83.36%)	4952 (72.05%)	635	455 (71.65%)	573 (90.23%)
memcached	133	110 (82.7%)	70 (52.63%)	48	43 (89.58%)	48 (100.0%)
node	20069	15161 (75.54%)	13911 (69.31%)	10502	4757 (45.29%)	8841 (84.18%)
geomean	1074.9	838.46 (77.99%)	726.89 (67.61%)	210.15	139.18 (66.22%)	185.68 (88.35%)

Table 3: The result for classification of callsites using our *type* policy on the O2 optimization level, when comparing to the ground truth obtained by our Clang/LLVM pass. The label perfect args denotes all occurrences when our result and the ground truth perfectly match regarding the required/provided arguments. The label perfect return denotes this for return values.

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. 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* schemes. **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 the 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 the 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 set up 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

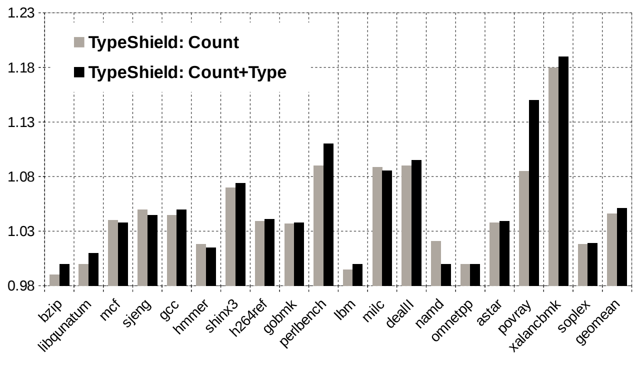


Figure 7: SPEC CPU2006 Benchmark Results.

Figure 7 depicts the runtime overhead obtained by applying our tool to several SPEC CPU2006 benchmarks in count mode and in count and type mode, respectively. The obtained geomean runtime overhead is under 1.1% performance drop when instrumenting using Dyninst. One reasons for the performance drop includes cache misses introduced by jumping between the old and the new executable section of the binary generated by duplicating and patching. This is necessary, because when outside of the compiler, it is nearly 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. Finally, our runtime overahead is lower (2%) than state-of-the-art tools (around 3%), thus qualifying TypeSchield as a highly practical tool.

O2 Target	AT	*count** limit (mean ± σ)			*count** median	*count* limit (mean ± σ)			*count* median	*type** limit (mean ± σ)			*type** median	*type* limit (mean ± σ)			*type* median
proftpd	396	330.31	±	48.07	343.0	334.5	±	51.26	311.0	310.58	±	60.33	323.0	337.41	±	54.09	336.0
pure-ftpd	13	5.5	±	4.82	6.5	9.87	±	4.32	13.0	4.37	±	4.92	2.0	8.12	±	4.11	7.0
vsftpd	10	7.14	±	1.81	6.0	7.85	±	1.39	7.0	5.42	±	0.95	6.0	6.42	±	0.96	7.0
lighttpd	63	27.75	±	10.73	24.0	41.19	±	13.22	41.0	25.1	±	8.98	24.0	41.42	±	14.29	38.0
mysqld	5896	2804.69	±	1064.83	2725.0	4281.71	±	1267.78	4403.0	2043.58	±	1091.05	1564.0	3617.51	±	1390.09	3792.0
postgres	2504	1964.83	±	618.28	2124.0	1990.59	±	574.53	2122.0	1747.22	±	727.08	2004.0	1624.07	±	707.58	1786.0
memcached	14	11.91	±	2.84	14.0	12.0	±	1.38	13.0	9.97	±	1.45	11.0	10.25	±	0.77	10.0
node	7230	3406.07	±	1666.9	2705.0	5306.05	±	1694.73	5429.0	2270.28	±	1720.32	1707.0	4229.22	±	2038.64	3864.0
geomean	216.61	129.77	±	43.99	127.62	166.09	±	40.28	171.97	105.13	±	38.68	92.74	144.06	±	38.38	141.82

Table 4: Restriction results of allowed callsites per calltarget for several test series on various targets compiled with Clang using optimization level O2. Note that the basic restriction to address taken only calltargets (see column AT) is present for each other series. The label *count denotes the best possible reduction using our *count* policy based on the ground truth collected by our Clang/LLVM pass, while *count* denotes the results of our implementation of the *count* policy derived from the binaries. The same applies to *type** and *type* regarding the *type* policy. A lower number of calltargets per callsite indicates better results. Note that our *type* policy is superior to the *count* policy, as it allows for a stronger reduction of allowed calltargets. We consider this a good result which further improves the state-of-the-art. Finally, we provide the median and the pair of mean and standard deviation to allow for a better comparison with other state-of-the-art tools.**

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 we encountered in our experiments is around 40% to 60% with Postgres having an increase of 150% in binary size. One cannot 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

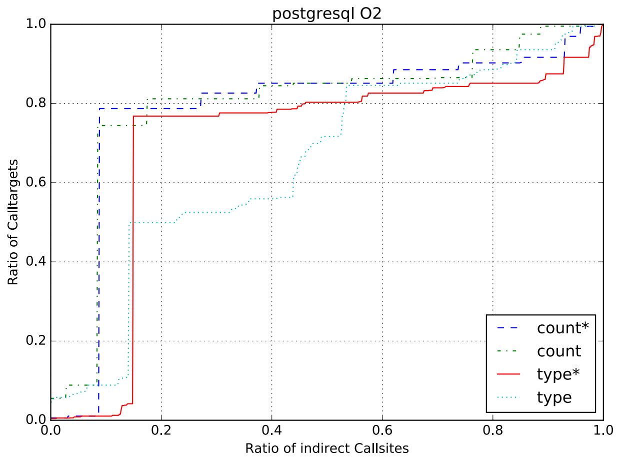


Figure 8: Postgresql -O2 CDF

Figures 8 depicts the CDF for the following program Postgresql when compiled with the -O2 Clang compiler flag. We selected this program randomly from our sample programs. 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 changes, the number of changes that can be seen within the type policies is vastly higher. The reason for that is fairly straightforward: 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 also relatively simple: the maximum number of calltargets a callsite can access has been reduced. Therefore, a lower number of calltargets is a higher percentage than before. However, we note that these results dependent on the particular structure of the hardened program.

6.6 Comparison with Other Tools

Target	AT	TypeArmor	IFCC	TypeShield (count)	TypeShield (type)
lighttpd	63	47	6	41	38
Memcached	14	14	1	13	10
ProFTPD	396	376	3	311	336
Pure-FTPD	13	4	0	13	7
vsftpd	10	12	1	7	7
PostgreSQL	2504	2304	12	2122	1786
MySQL	5896	3698	150	4403	3792
Node.js	7230	4714	341	5429	3864
geomean	216.6	162.1	7.6	172.0	141.8

Table 5: The medians of calltargets per callsite for different restriction methods. Note that the smaller the geomean numbers are, the better the technique is. AT is a technique which allows calltargets that are address taken. IFCC is a compiler based solution and depicted here as a reference for what is possible when source code is available. TypeArmor and TypeShield on the other hand are binary-based tools. We can see that our type-based tool on average reduces the number of calltargets by 35% when compared to AT method and by 13% when comparing with TypeArmor. Finally, we are think that by tweaking of our analysis we further can reduce the calltarget set for each callsite.

Table 5 depicts a comparison between TypeShield, TypeArmor and IFCC with respect to 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 implement 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 with respect to precision. While we anticipate that it is possible to surpass TypeArmor implementing those two solutions in our tool, we deem it nearly impossible to be able to compete with IFCC, which can directly operate on the source code level and having 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, where non-returning functions like exit are not detected reliably in some cases. As a result, we cannot test the Pure-FTP 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 misclassifications for both calltargets and callsites.

Second, TypeShield draws 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, otherwise, TypeShield is equivalent to a parameter count-based implementation. Occurrences of such situations are quite rare, as we learned with our experiments. With a study based on source level information, we could not detect a difference between our *type* policy and a *count* policy. However, when using our tool, we were able to detect differences, which reinforce 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 [11]. For this reason, we assume that TypeShield runs side by side with an ideal backward-edge protection mechanism such as a shadow [9]. However, the main goal of TypeShield is to complement shadow stack based defenses which fails to account for attacks not violating 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 [26]. However, TypeShield is highly useful in situations where the source code is typically not available (*e.g.,* off-the-shelf programs), where programs rely on many libraries, and where the recompilation of all the shared libraries is not possible. Further, binary based tools as TypeShield can offer precise protection when source code is not available or recompilation is not feasible or desirable.

Finally, while a major step forward, TypeShield cannot thwart all possible attacks, as even solutions with access to source code are unable to protect against all possible attacks [8]. In contrast, 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 similar tools. Moreover, TypeShield provides a strong mitigation for other types of code-reuse attacks as well.

8 RELATED WORK

Recursive-COOP [10], COOP [25], Subversive-C [21] and the attack of Lan *et al.* [19] are forward-edge based CRAs which cannot be addressed with: *i)* shadow stacks techniques and hardware-based approaches such as Intel CET [4] (*i.e.,* since advanced COOP do not violate the caller/callee convention), *ii)* coarse-grained Control-Flow Integrity (CFI) [5, 6] techniques, and *iii)* 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.

The following tools address vTable protection through binary instrumentation, but fail to mitigate against COOP: vfGuard [23], and vTint [29]. The only binary-based tool which we are aware of that can protect against COOP is TypeArmor [27]. TypeArmor uses a fine-grained CFI policy based on caller/callee (but only indirect callsites) matching, which checks during runtime if the number of provided and needed parameters match.

TypeShield is related to TypeArmor [27], since we also enforce strong binary-level invariants on the number of function parameters. Further, TypeShield also aims for 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 width of their types. This results in much smaller buckets that in turn can only target a smaller subset of all address-taken functions.

9 CONCLUSION

In this paper, we presented TypeShield, a runtime fine-grained CFI-enforcing tool which operates on program binaries. Our tool precisely and efficiently filters legitimate from illegitimate forward indirect control flow transfers by using a novel runtime type-checking technique based on function parameter type-checking and parameter-counting.

We have implemented TypeShield and applied it to real software such as web servers, FTP servers and the SPEC CPU2006 benchmark. We demonstrated through extensive experiments that TypeShield has higher precision w.r.t. the calltarget set per callsite than existing state-of-the-art tools, while maintaining a lower runtime ($\leq 2\%$) performance overhead. To date, we were able to improve on parameter count based techniques by reducing the possible calltargets per callsite ratio by 35% with an overall reduction of more than 13% when comparing with similar state-of-the-art tools. Next to a more precise analysis, the tangible outcome is a considerably reduced attack surface which can be further be improved by tweaking our analysis. Finally, in the spirit of open research, we have made the source code of TypeShield and the evaluation results publicly available, thus we support reproducibility in this fast-moving research field by providing comprehensive descriptions of our experiments.

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] 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)*.
[6] 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)*.
[7] 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)*.
[8] 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)*.
[9] 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)*.
[10] 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)*.
[11] 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)*.
[12] Mohamed Elsabagh, Fleck Dan, and Angelos Stavrou. 2017. Strict Virtual Call Integrity Checking for C ++ Binaries. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.
[13] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskosr. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[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] Wayne Jansen. 2009. Directions in Security Metrics Research. In *NISTIR 7564*. http://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7564.pdf.
[17] I. JTC1/SC22WG21. 2013. ISO/IEC 14882:2013 Programming Language C++ (N3690). https://isocpp.org/files/papers/N3690.pdf.
[18] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2009. *Data flow analysis: Theory and Practice*. CRC Press.
[19] 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*.
[20] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the USENIX Conference on Security (USENIX SEC)*.
[21] 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)*.
[22] 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.
[23] 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)*.
[24] G. Ramalingam. 1994. The Undecidability of Aliasing. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*.
[25] 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)*.
[26] 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)*.
[27] 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)*,.
[28] 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)*.
[29] 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)*.
[30] 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)*.
[31] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the USENIX conference on Security (USENIX SEC)*.

APPENDIX

10 EXTENDED BACKGROUND

10.1 Polymorphism in C++ Programs

Polymorphism, along inheritance and encapsulation, are the most used modern object-oriented concepts in C++. In C++, polymorphism allows accessing different types of objects through a common base class. A pointer of the type of the base object can be used to point to object(s) which are derived from the base class. In C++, there are several types of polymorphism: *a)* compile-time (or static, usually is implemented with templates), *b)* runtime (dynamic, is implemented with inheritance and virtual functions), *c)* ad-hoc (*e.g.,* if the range of actual types that can be used is finite and the combinations must be individually specified prior to use), and *d)* parametric (*e.g.,* if code is written without mention of any specific type and thus can be used transparently with any number of new types). The first two are implemented through early and late binding, respectively. In C++, overloading concepts fall under the category of *c)* and virtual functions, templates or parametric classes fall under the category of pure polymorphism. However, C++ provides polymorphism through: *i)* virtual functions, *ii)* function name overloading, and *iii)* operator overloading. In this paper, we are concerned with dynamic polymorphism, based on virtual functions (10.3 and 11.5 in ISO/IEC N3690 [17]), because it can be exploited to call: *x)* illegitimate virtual table entries (not) contained in the class hierarchy by varying or not the number of parameters and types, *y)* legitimate virtual table entries (not) contained in the class hierarchy by varying or not the number of parameters and types, and *z)* fake virtual tables entries not contained in the class hierarchy by varying or not the number of parameters and types. By legitimate and illegitimate virtual table entries we mean those virtual table entries which for a single indirect callsite lie in the virtual table hierarchy. More precisely, a virtual table entry is legitimate for a callsite if from the callsite to the virtual table containing the entry there is an inheritance path (see [14]). Virtual functions have several uses and issues associated, but for the scope of this paper we will look at the indirect callsites which are exploited by calling illegitimate virtual table entries (*i.e.,* functions) with varying number and type of parameters, *x)*. More precisely, *1) load-time enforcement:* as calling each indirect callsite (*i.e.,* callee) requires a fixed number of parameters which are passed each time the caller is calling, we enforce a fine-grained CFI policy by statically determining the number and types of all function parameter that belong to an indirect callsite, and *2) runtime verification:* as differentiating during runtime legitimate from illegitimate indirect caller/callee pairs requires parameter type (along parameter number), we check during run-time before each indirect callsite if the caller matches with the callee based on the previously added checks.

11 SECURITY ANALYSIS

11.1 Checking Indirect Calls in Practice

To the best of our knowledge, only the IFCC/VTV [26] tools (up to 8.7% performance overhead) are deployed in practice which can be used to check legitimate from illegitimate indirect forward-edge calls during runtime. vPointers are checked based on the class hierarchy. Furthermore, ShrinkWrap [14] (to the best of our knowledge not deployed in practice) is a tool which further reduces the legitimate virtual table ranges for a given indirect callsite through precise analysis of the program class hierarchy and virtual table hierarchy. Evaluation results show similar performance overhead but more precision with respect to legitimate virtual table entries per callsite. We noticed by analyzing the previous research results that the overhead incurred by these security checks can be very high due to the fact that for each callsite many range checks have to be performed during runtime. Therefore, in our opinion, despite its security benefit these types of checks cannot be applied to high performance applications.

A number of other highly promising tools (albeit also not deployed in practice) can overcome some of the drawbacks of the previously described tools. Bounov *et al.* [7] presented a tool ($\approx$ 1% runtime overhead) for indirect forward-edge callsite checking based on virtual table layout interleaving. The tool has better performance than VTV and better precision with respect to allowed virtual tables per indirect callsite. Its precision (selecting legitimate virtual tables for each callsite) compared to ShrinkWrap is lower since it does not consider virtual table inheritance paths. vTrust [28] (average runtime overhead 2.2%) enforces two layers of defense (virtual function type enforcement and virtual table pointer sanitization) against virtual table corruption, injection and reuse. TypeArmor [27] ($\leq$ than 3 % runtime overhead) enforces a CFI-policy based on runtime checking of caller/callee pairs and function parameter count matching. It is important to note that there are no C++ language semantics which can be used to enforce type and parameter count matching for indirect caller/callee pairs, this could be addressed with specifically intended language constructs in the future.

11.2 Security Implications of Indirect Calls

The C++ language standard (12.7 [17]) does not specify what happens when calling different virtual table entries from an indirect callsite. The standard says that we have a virtual function-related undefined behavior when: *a virtual function call uses an explicit class member access and the object expression refers to the complete object of x or one of that object's base class subobjects but not x or one of its base class subobjects.* As undefined behavior is not a clearly defined concept, we argue that in order to be able to deal with undefined behavior or unspecified behavior related to virtual function calls one needs to know how these language-dependent concepts are implemented inside the used compilers.

Forbidden forward-edge indirect calls are the result of a vPointer corruption. A vPointer corruption is not a vulnerability, but rather a capability which can be the result of a spatial or temporal memory corruption triggered by: (1) bad-casting [20] of C++ objects, (2) buffer overflow in a buffer adjacent to a C++ object or a use-after-free condition [25]. A vPointer corruption can be exploited in several ways. A manipulated vPointer can be exploited by pointing it in any existing or added program virtual table entry or into a fake virtual table which was added by an attacker. For example in case a vPointer was corrupted than the attacker could highjack the control flow of the program and start a COOP attack [25].

vPointer corruptions are a real security threat which can be exploited if there is a memory corruption (*e.g.,* buffer overflow) which is adjacent to the C++ object or a use-after-free condition. As a consequence, each corruption which can reach an object (*e.g.,* bad object casts) is a potential exploit vector for a vPointer corruption. Interestingly to notice in this context is that through: (1) memory layout analysis (through highly configurable compiler tool chains) of source code based locations which are highly prone to memory corruptions such as declarations and uses of buffers, integers or pointer deallocations one can obtain the internal machine code layout representation. (2) analysis of a code corruption which is adjacent (based on (1)) to a C++ object based on application class hierarchy, the virtual table hierarchy and each location in source code where an object is declared and used (*e.g.,* modern compiler tool chains can spill out this information for free), one can derive an analysis which can determine—up to a certain extent—if a memory corruption can influence (*e.g.,* is adjacent) to a C++ object.

Finally, tools based on these two concepts (*i.e.,* (1) and (2)) can be used by attackers, *e.g.,* to find new vulnerabilities, and by defenders to harden the source code only at the places which are most exposed to such vulnerabilities (*i.e.,* targeted security hardening).

11.3 Imprecise Parameter-Count Policies

TypeArmor [27] is a tool that can enforce a CFI runtime policy for dispatching of callsites based only on parameter count. The authors argue that their policy reports only an *overestimation* for the parameters prepared by a callsite and *underestimation* for the number of parameters consumed by the matching calltargets. The authors suggest that their technique is effective against COOP attacks.

We do not fully agree with this claim and, furthermore, we believe that their callsite vs. calltarget set enforcing policy is too permissive and thus many potential indirect forward edge based control flow transfers are possible. Consider the following example. In the best case for each callsite preparing, say, $p = 4 \in [1, 6]$ parameters their policy could theoretically allow only the calltargets which consume the same number as parameters as prepared, $c = 4 \in [1, 6]$. Note that this does not hold due to the aforementioned callsite overestimation and calltarget underestimation, thus all possible numerical mismatches are allowed by their policy as long as p is greater or equal to c.

- TypeArmor ***ideally*** would allow for a single callsite a set of calltargets containing a maximum of 117649 possibilities if we consider the maximum value of provided parameters to be $p = 6$ (due to $p \in [1,6]$ possible provided parameters). Now, consider 7 C++ integer parameter types t: *int*, *char*, *unsignedchar*, *bool*, *long*, *unsignedlong*, and *short*. Thus, we obtain $t^p = 7^6 = 117649$ allowed calltargets per callsite if TypeArmor is used. Note that for simplicity reasons we considered $t = 7$ but in practice t is often even larger since there are many types of parameters in C++. The complete list of fundamental C++ types contains 20 types; not including data structures or object types. Thus, all these data types would be ignored by TypeArmor. Also, note that all other callsites having more than 6 parameters would be not checked by TypeArmor as well.

- TypeArmor ***actually*** allows more than t^p calltargets per callsite. If we have $t = 7$ integer types due to TypeArmors overestimation and underestimation we get for each callsite an additional number of calltargets. Let $p = 6$, then we get $c = 6x + 5y + 4z + 3t + 2p + 1v$ where: x is the sum of all calltargets consuming 6 parameters, y is the sum of all calltargets consuming 5 parameters and so on down to 0 parameters. Note that this holds since TypeArmor allows more parameters to be provided than consumed by the calltarget. Then, $c = 2100 = 600 + 500 + 400 + 300 + 200 + 100 \ iff \ x = y = z = t = p = v = 100$. Note that $x = 100$ is feasible under realistic conditions in large applications (*i.e.,* Google Chrome, Firefox). Next 2100 is added to 7^6. Thus, for a single callsite providing $p = 6$ parameters TypeArmor allows theoretically in total $7^6 + 2100 = 1197496$ calltargets for each callsite. Similar reasoning applies to $p = 5$ where we get $7^5 + (1500 = 500 + 400 + 300 + 200 + 100) = 18307 \ iff \ x = y = z = t = p = v = 100$ allowed calltarget per callsite, or $p \in [1,4]$, too.

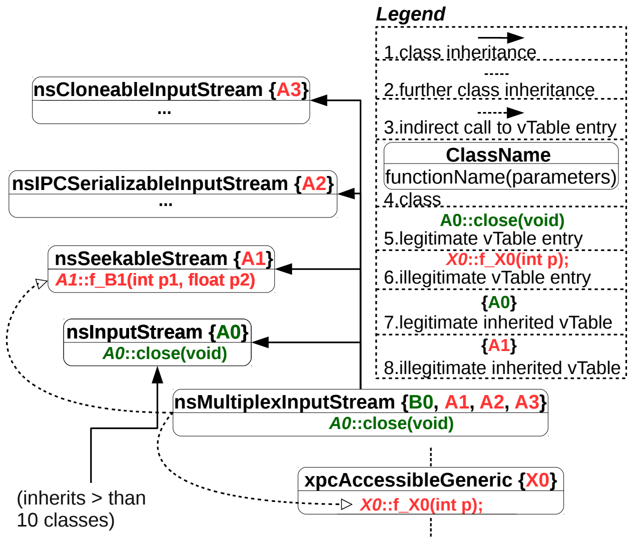
Finally, as TypeArmor is too permissive we present TypeShield which deals with the variable type state explosion due to different parameter types by considering an approximation (note that alias analysis and thus type analysis in binaries is undecidable [24]) of parameter types based on register width. Consequently, the allowed calltarget set for each callsite is drastically reduced.

11.4 Real COOP Attack Example

Figure 9 depicts the COOP attack example used as proof of concept exploit presented in [25] and to perform a COOP attack on the Firefox browser. A buffer overflow bug was used in order to call into existing virtual table entries by using a main loop gadget. The attack concludes with the opening of a Unix shell.

A real-world bug, CVE-2014-3176, was exploited by Crane *et al.* [10] in order to perform another COOP attack, on the Chromium browser. The details of the second attack are highly complex (*i.e.,* involving not properly handled interaction of extensions, IPC, the sync API, and Google V8) and for this reason we only briefly present the first documented COOP exploit on a Linux machine.

The C++ class `nsMultiplexInputStream` contains a main loop gadget inside the `nsMultiplexInputStream::Close(void)` function which is performing indirect calls by dispatching indirect calls on the objects contained in the array. The objects contained in the array during normal execution are of type `nsInputStream` and each of the objects will call the `Close(void)` function in order to close each of the previously opened streams. For performing the COOP attack, the attacker crafts a C++ program containing



Legend
1.class inheritance
2.further class inheritance
3.indirect call to vTable entry
ClassName
functionName(parameters)
4.class
A0::close(void)
5.legitimate vTable entry
X0::f_X0(int p);
6.illegitimate vTable entry
{A0}
7.legitimate inherited vTable
{A1}
8.illegitimate inherited vTable
nsCloneableInputStream {A3}
...
nsIPCSerializableInputStream {A2}
...
nsSeekableStream {A1}
A1::f_B1(int p1, float p2)
nsInputStream {A0}
A0::close(void)
nsMultiplexInputStream {B0, A1, A2, A3}
A0::close(void)
xpcAccessibleGeneric {X0}
X0::f_X0(int p);
(inherits > than 10 classes)

Figure 9: Class inheritance hierarchy of the classes involved in the COOP attack against the Firefox browser. Red letters indicate forbidden virtual table entries and green letters indicate allowed virtual table entries for the given indirect callsite contained in the main loop gadget.

an array buffer holding six fake objects. These fake objects can call inside (and outside) the initial class and virtual table hierarchies with no constraints. During the attack a buffer is created in order to hold the fake objects. The crafted buffer will be used instead of the real code in order to call different functions available in the program code. For example, the attacker calls a function contained in the class `xpcAccessibleGeneric` which is not in the class hierarchy or virtual table hierarchy of the initially intended type of objects used inside the array. Moreover, the header file of this class (`xpcAccessibleGeneric`) is not included in the class `nsMultiplex-InputStream`. In total six fake objects are used to call into functions residing in unrelated class hierarchies with varying number of parameters and return types. The final goal of this attack is to prepare the program memory such that a Unix shell can be opened at the end of this attack.

This example illustrates why detecting vPointer corruptions is not trivial for real-world applications. As depicted in Figure 9, the class `nsInputStream` has 11 classes which inherit directly or indirectly from this class. The classes `nsSeekableStream`, `nsIPCSerializableInputStream` and `nsCloneableInputStream` provide additional inherited virtual tables which represent illegitimate calltargets for the initial `nsInputStream` objects and legitimate calltargets for the six fake objects which were added during the attack. Furthermore, declaration and usage of the objects can be widely spread out in the source code. This makes detection of the object types (*i.e.,* base class), range of virtual tables (*i.e.,* longest virtual table inheritance path for a particular callsite) and parameter types of the virtual table entries (*i.e.,* functions) in which it is allowed to call a trivial task for source code applications, but a hard task when one wants to apply similar security policies (*e.g.,* which rely on parameter types of virtual table entries) to binary executables.

11.5 Variadic Function Example

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