TYPESHIELD: Precise Protection of Forward Indirect Calls in Binaries

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

High security, high performance and high availability applications are usually implemented in C/C++ for modularity, performance and compatibility to name just a few reasons. Virtual functions, which facilitate late binding, are a key ingredient in facilitating runtime polymorphism in C++ because it allows and object to use general (its own) or specific functions (inherited) contained in the class hierarchy. Despite the alarmingly high number of *vptr* corruption vulnerabilities, the *vptr* corruption problem has not been sufficiently addressed by researchers.

In this paper, we present *TypeShield*, a runtime *vptr* corruption detection tool. It is based on instrumentation of executables at load time and uses a novel runtime type and function parameter counter technique in order to overcome the limitations of current approaches and efficiently verify dynamic dispatching during runtime. In particular, *TypeShield* can be automatically and easily used in conjunction with legacy applications or where source code is missing. It achieves higher caller/caller matching (precision) and with reasonable runtime overhead. We have applied *TypeShield* to web servers, FTP servers and SPEC CPU2006 benchmark and were able to efficiently and with low performance overhead protect these applications from forward indirect edge *vptr* based corruptions. Our evaluation shows that our target reduction schema achieves an additional reduction of the possible calltargets per callsite of up to 20% with an overall reduction of about 9% when comparing with other state-of-the-art parameter count based approaches.

1 Introduction

Control-Flow Integrity (CFI) [**?**, **?**] is one of the most used techniques to secure program execution flows against advanced Code-Reuse Attacks (CRAs). Advanced CRAs such as the recently published COOP [**?**] and its extension [**?**] or the attacks described by the Control Flow Bending paper [**?**] are able to bypass most traditional CFI solutions, as they focus on indirect callsites, which are not as easy to determine at compile time.

This is a problem for applications written in C++, as one of its principle is inheritance and virtual functions. The concept of virtual functions allows the programmer to overwrite a virtual function of the base-class with his own implementation. While this allows for much more flexible code, this flexibility is the reason COOP actually works. The problem is that in order to implement virtual functions, the compiler needs to generate a table of all virtual functions for each class containing them and provide to each instance of such a class a pointer to said table. COOP now leverages a memory corruption to inject their own object with a fake virtual pointer, which basically gives him control over the whole program, while the control flow still looks genuine, as no code was replaced.

Current solutions. There exist several source code based solutions that either insert run-time checks during the compilation of the program like SafeDispatch [**?**], ShrinkWrap [**?**] or IFCC/VTV [**?**], which is the solution it is based on. Others modify and reorder the contents of the virtual table as their main aspect like the paper by Bounov et al. [**?**]. While the recently published Redactor++ [**?**] implements a combination of those ideas.

While this might seem that only C++ is vulnerable, while C is safe, this notion is wrong, as the Control Flow Bending paper [**?**] proposes attacks on nginx leveraging global function pointers, which are used to provide configurable behavior.

As previously mentioned, there exist many solutions when one tries to tackle this problem while access to the application in question is provided. However, when we are faced with proprietary third party binaries, which are provided as is and without the actual source code, the number of tools that can protect against COOP or similar attacks is rather low.

Limitations. TypeArmor [**?**] implements a fine grained forward edge CFI policy based on parameter count for binaries. It calculates invariants for calltargets and indirect callsites based on the number of parameters they use by leveraging static analysis of the binary, which then is patched to enforce those invariants during run-time. The main shortcoming of TypeArmor is that even with high precision in the classification of calltargets and callsites, one cannot exclude calltargets with lower parameter number from callsites, for one due compatibility and also due to variadic functions, which are a special case in themselves. This basically means that when a callsite prepares 6 parameters, it is able to call all ad-

dress taken functions. This generates a considerable attack surface due to the many situations in which this policy can be naturally circumvented.

In this paper, we present TYPESHIELD, a runtime illegitimate forward calls detection tool that can be seamlessly integrated with large scale applications such as web servers. It takes the binary of a program as input and it can automatically instrument the binary in order to detect illegitimate indirect calls at runtime. We implemented TYPESHIELD to demonstrate a possible remedy of this problem by introducing parameter types into the classification of callsites and calltargets. We explore to what extent we can further narrow down the set of possible targets for indirect callsites and manage to stop the exploitation at the binary level w.r.t. TypeArmor. Our conclusion is that our tool can not stop all possible attacks since even solutions with access to source code are unable to protect against all possible attacks [**?**]. Nevertheless, we show that TYPESHIELD, our binary based tool can stop all COOP attacks published to date and significantly raises the bar for an adversary when compared our tool with TypeArmor and other similar tools. Moreover, TYPESHIELD provides strong mitigation for many types of code-reuse attacks (CRAs) for program binaries, without the need for source code.

Our Insight. TYPESHIELD is based on a forward-edge CFI policy that relies on a precise construction of both the callee prototypes and callsite signatures and then uses this information to enforce that each callsite targets matching functions only. For example, TYPESHIELD disallows an indirect call 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. Additionally, TYPESHIELD incorporates an improved protection policy which further reduces the possible target set of callees for each callsite. Our novel policy is based on the insight that if the binary adhere to the standard calling convention for indirect calls, undefined arguments at the callsite are not used by any callee by design. TYPESHIELD trashes these so-called spurious arguments and thus breaks all published COOP and improved COOP-like exploits. These exploits all chain virtual method calls that disrespect calling conventions to achieve convenient data flows between gadgets [**?**].

Current binary based techniques enforce imprecise forward-edge CFI policies, often allowing controll transfers from any valid callsite to any valid referenced entry point [**?**, **?**]. In the best case, existing policies only reduce the target set by removing all entry points of other modules unless they were explicitly exported or observed at runtime [**?**]. In contrast, TYPESHIELD matches up indirect callsites with a more precise target set in a considerably more precise many-to-many relationship than TypeArmor. It is based on a use-def analysis at all possible callees to approximated the function prototypes, and liveness analysis at indirect callsites to approximate callsite signatures. This efficiently leads to a more precise CFG of the binary program in question, which could be used by other systems in order to gain more a precise CFG on which to enforce their policies.

TYPESHIELD can protect only forward indirect edges at the binary level. Previous research has shown that, a backward-edge protection such as a shadow stack [**?**] or context-sensitive CFI [**?**] is still essential to ensure the integrity of return addresses at runtime [**?**]. In this paper, we assume an ideal backward-edge protection mechanism such as a shadow stack with no design faults [**?**]. TYPESHIELD complements such backward-edge defenses by addressing attacks that take place without violating the integrity of the return path. Specifically, TYPESHIELD provides a precise protection against COOP exploits as well as improved COOP derivations [**?**, **?**, **?**, **?**].

TYPESHIELD is not more precise than source code based approaches such as IFCC/VTV [**?**]. IFCC/VTV are strong compiler based defenses which produce binaries which can resist control-flow hijacking attacks. It is well known that source-code based techniques are more precise when enforcing fine-grained policies based on program constructs (such as the C++ vTable hierarchy or generic data types) for mitigation purposes. However, there are still important reasons to study and improve binary-level defenses. First, the source code for many off-the-shelf programs is not always available. Second, real-world programs rely on a plethora of shared libraries and recompiling all shared libraries is not always possible. This is true even for purely open-source projects. Third, even if the source code of the libraries is available, recompiling big projects with dynamic dependencies is, again, a demanding task. Even state-of-the art defenses that enforce CFI policies at the source level such as Interleaving [**?**] do not support dynamic libraries. Notice that mixing CFI-protected with non-protected code is impossible since applying CFI only on a part of the CFG would crash legitimate executions. In contrast, with a binary-level solution, we can offer strong protection even if the source code is not available or when recompilation is not feasible (or desirable).

Contributions. In summary, we make the following contributions:

- **Security analysis of forward indirect calls.** We analyzed the usage of illegitimate indirect forward calls in detail, thus providing security researchers and practitioners a better understanding of this emerging attack vector.

- **Illegitimate indirect calls detection tool.** We designed and implemented TYPESCHIELD, 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. An open-source implementation of TYPESCHIELD is available at https://github.com/tba/typeshield.

- **Experiments.** We demonstrate trough extensive experiments that our precise binary-level CFI strategy can mitigate advanced code reuse attacks in absence of C++ semantics. For example TYPESCHIELD can protect against COOP [**?**] and its currently published variations [**?**, **?**, **?**, **?**].

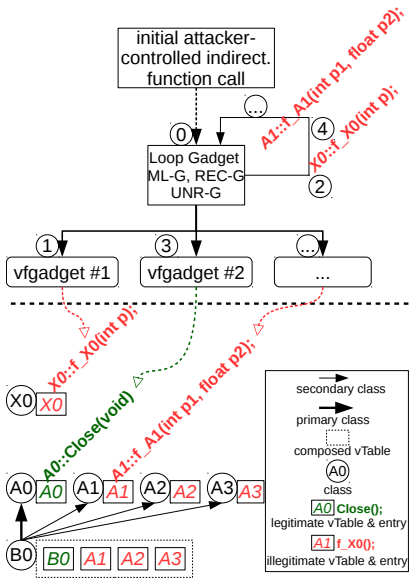

Figure 1: Code presenting how a COOP loop gadget works.

2 C++ Forbidden Forward Calls Exposed

Polymorphism in C++. Polymorphism along inheritance and encapsulation are the most used modern object-oriented concepts in C++. Polymorphism in C++ 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 it is called parametric polymorphism). 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. C++ provides polymorphism through: *i)* virtual functions, *ii)* function name overloading, and *iii)* operator overloading. In this paper, we will be concerned with dynamic polymorphism—based on virtual functions (10.3 and 11.5 in ISO/IEC N3690 [**?**])—because these can be exploited to call: *x)* illegitimate vTable entries not/contained in the class hierarchy by varying or not the number of parameters and types, *y)* legitimate vTable entries not/contained in the class hierarchy by varying or not the number of parameters and types, and *z)* fake vTables entries not contained in the class hierarchy by varying or not the number of parameters and types. By legitimate and illegitimate vTable entries we mean those vTable entries which for a single indirect callsite lie in the vTable hierarchy. More precisely, a vTable entry is legitimate for a callsite if from the callsite to the vTable containing the entry there is an inheritance path (see [**?**]). Virtual functions have several uses and issues associated, but for the scope of this paper we will look at the indirect callsites which are exploiter by calling illegitimate vTable entries (functions) with varying number and type of parameters, *x)*. More precisely, *1)* load-time enforcement: as calling each indirect callsite (callee) requires a fix 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. *2)* runtime verification: as checking during runtime legitimate from illegitimate indirect caller/callee pairs requires parameter type (along parameter number), we check during run-rime before each indirect callsite if the caller matches to the callee based on the previously added checks.

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

Checking Indirect Forward-Edge Calls in Practice. As far as we know, there is only the IFCC/VTV [**?**] tools (up to 8.7% performance overhead) deployed in practice which can be used to check legitimate from illegitimate indirect forward-edge calls during compile time. vPointers are checked based on the class hierarchy. ShrinkWrap [**?**] (as far as we know not deployed in practice) is a tool which further reduces the legitimate vTbles ranges for a given indirect callsite through precise analysis of the program class hierarchy and vTable hierarchy. Evaluation results show similar performance overhead but more precision w.r.t. to legitimate vTables 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, despite its security benefit these types of checks can not be applied in our opinion to high performance applications.

As alternative, there are other highly promising tools (not deployed in practice) that can be used to mitigate some of the drawbacks of the previous tools. Bounov et al [**?**] presented a tool ($\approx$ 1% runtime overhead) for indirect forward-edge callsite checking based on vTable layout interleaving. The tool has better performance than VTV and better precision w.r.t. allowed vTables per indirect callsite. Its precision (selecting legitimate vTables for each callsite) compared to ShrinkWrap is lower since it does not consider vTable inheritance paths. vTrust [**?**] (average runtime overhead 2.2%) enforces two layers of defense (virtual function type enforcement and vTable pointer sanitization) against vTable corruption, injection and reuse. TypeArmor [**?**] ($\leq$ than 3 % runtime overhead) enforces an CFI policy based on runtime checking of caller/caller pairs based on function parameter count matching (coarse grained, parameter types and more than six parameters can be used as well). Important to notice is that there are no C++ language semantics which can be used to enforce type and parameter count matching for indirect call/-callee pairs, this could be addresses with specifically intended language constructs in the future.

Security Implications of Forbidden Indirect Calls. The C++ language standard (12.7 [**?**]) does not specify what happens when calling different vTable entries from an indirect callsite. The standard says that we have 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 languages 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 through: (1) bad-casting [**?**] of C++ objects, (2) buffer overflow in a buffer adjacent to a C++ object or a use-after-free condition [**?**]. A vPointer corruption can be exploited in several ways. A manipulated vPointer can be exploited by pointing it in any existing or added program vTable entry or into a fake vTable 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 [**?**].

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 vTble 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 (is adjacent) to a C++ object.

Finally, we notice that by building tools based on this two concepts (i.e., (1) and (2)) attackers (e.g., used to find new vulnerabilities) and for defenders which can harden the source code with checks only at the places which are most exposed to such vulnerabilities (i.e., we name this targeted security hardening).

Real COOP Attack Example. The given example depicted in Figure 2 is a proof of concept exploit extracted from [**?**] and used in order to perform a COOP attack on the Firefox browser. A buffer overflow bug was used in order to call into existing vTable entries by using a main loop gadget. The attack concludes with opening of an Unix shell. A real-world bug, CVE-2014-3176, was exploited by Crane et al. [**?**] in order to perform another COOP attack, on the Chromium browser. The details of the second attack are far to complex (i.e., involves not properly handled interaction of extensions, IPC, the sync API, and Google V8) and for this reason we briefly present the first documented COOP exploit on a Linux machine.

The C++ class `nsMultiplexInputStream` contains a main loop gadget inside the function `nsMultiplexInputStream::Close(void)` which is performing an 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.

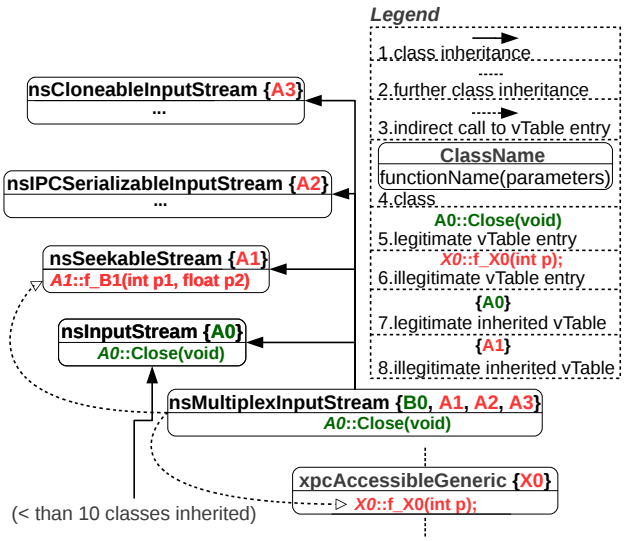

Figure 2: Class inheritance hierarchy of the classes involved in the COOP attack against the Firefox browser. Red letters indicate forbidden vTble entries and green letters indicate allowed vTable entries for the given indirect callsite contained in the main loop gadget.

In order to perform the COOP attack the attacker crafts a C++ program containing an array buffer holding six fake objects. Fake objects can call inside (and outside) the initial class and vTable hierarchies with no constraints. During the attack a buffer is created in order to hold the fake objects. The crafted buffer will be called 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 vTable 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 `nsMultiplexInputStream`. In total six fake objects are used to call into functions residing in not related 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 2 the class `nsInputStream` has 11 classes which inherit directly or indirectly from this class. The classes `nsSeekableStream`, `nsIPCSerializableInputStream` and `nsCloneableInputStream` provide additional inherited vTables 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 wide spread in the source code. This makes detection of the object types (base class), range of vTables (longest vTable inheritance path for a particular callsite) and parameter types of the vTable entries (functions) in which it is allowed to call a trivial task for source code (i.e., current research work is mostly concerned with performance issues) applications but a hard task in our opinion when one wants to apply similar security policies (e.g., which rely on parameter types of vTable entries) to executables.

3 Overview

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

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

TypeShield Impact on COOP. The problem with relying solely on the parameter count is that a callsite can use any call-target as long as the parameter count requirement is fulfilled, even if the parameter types do not match (imagine 8-bit values provided but 64-bit values required). Therefore, we extend the classification schema to the parameter types: *1)* indirect callsites provides a maximum wideness to each parameter (possibly overestimated compared to source), and

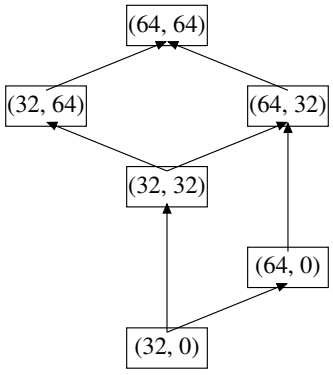

Figure 3: Example for the wideness based schema when only using a parameter wideness of 64-, 32- and 0-bit and only two parameters.

2) calltargets require a minimum wideness for each parameter (possibly underestimated compared to source).

The basic idea stays the same, the provision must be no lower than the requirement. However, the approach is more fine-grained applying to the wideness of each parameter. The result is that we split the buckets of TypeArmor up into smaller ones as shown in the limited example depicted in Figure 3.

4 Design

In this section, we cover the design of TYPESHIELD. We first present the details of the *count* policy in §4.1—as introduced by [**?**]—and the new *type* policy in §4.2. Then we present the theory needed to transform set-based analysis to register based ones in §4.3. Finally, we present the implementation details of the calltarget analysis in §4.4 and callsite analysis in §4.5 for each of our policies.

4.1 *Count* Policy

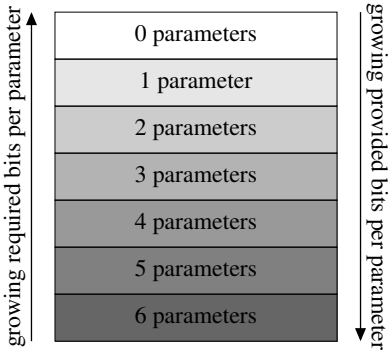

Figure 4: *Count* policy classification schema for callsites and calltargets.

What we call the *count* policy is essentially the policy introduced by TypeArmor [**?**]. 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 4.

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

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

4.2 *Type* Policy

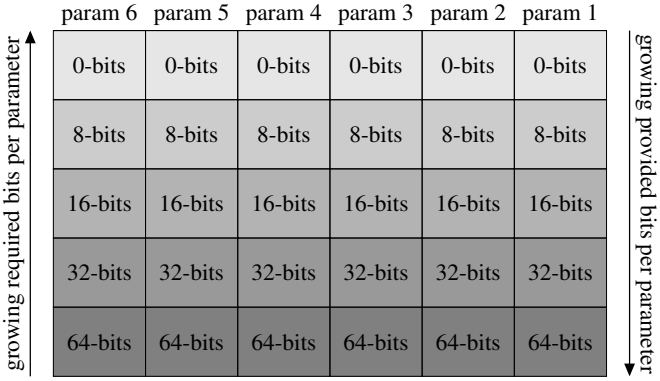

Figure 5: *Type* policy schema for callsites and calltargets.

What we call the *type* policy is the idea of not only relying on the parameter count but also on the parameter type. However, due to complexity reasons, we are restricting ourselves to the general purpose registers, which the SystemV ABI designates as parameter registers. Furthermore, we are not inferring the actual type of the data but the wideness of the data stored in the register. The schema again is that we have calltargets requiring wideness and the callsite providing it as depicted in Figure 5.

We are currently interested in x86-64 binaries, the registers we are looking at are 64-bit registers that can be accessed in four different ways: *1)* the whole 64-bits of the register, meaning a wideness of 64, *2)* the lower 32-bits of the register, meaning a wideness of 32, *3)* the lower 16-bits of the register, meaning a wideness of 16, and *4)* the lower 8-bits of the register, meaning a wideness of 8.

Four of those registers can also directly access the higher 8-bits of the lower 16-bits of the register. For our purpose we register this access as a 16-bit access. Based on this information, we can assign a register one of 5 possible types $\mathscr{T} = \{64, 32, 16, 8, 0\}$. We also included the type 0 to model the absence of data within a register. Similar to the *count* policy, we allow overestimation of types in callsites and underestimation of types in calltargets. However, the matching idea is different, because as can we depict in Figure 5, the type of a calltarget and a callsite no longer depends solely on its parameter count, each callsite and calltarget has its type

from the set of $\mathscr{T}^6$, with the following comparison operator: $u \leq_{type} v :\Longleftrightarrow \forall_{i=0}^{5} u_i \leq v_i$, with $u, v \in \mathscr{T}^6$

Again we allow any callsite *cs* call any calltarget *ct*, when it fulfills the requirement $ct \leq cs$. Meaning, just having an equal or lesser number of parameters than a callsite, does no longer allow a calltarget being called there, thus restricting the number of calltargets per callsite even further. A function that requires 64-bit in its first parameter, and 0-bit in all other parameters, would have been callable by a callsite providing 8-bit in its first and second parameter when using the *count* policy, however in the *type* policy this is no longer possible. Thus, it should decrease the number of targets per bucket.

4.3 Instruction Analysis

Usually data-flow analysis algorithms are based on set of variable or sets of definitions, which both are basically unbounded. However, we are analyzing the state of registers, which are baked into hardware and therefore their number is given, thus requiring us to adapt the data-flow theory to work on tuples.

The set $\mathscr{I}$ 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 $i \in \mathscr{I}$ can non-exclusively perform two kinds of operations on any number of existing registers: *1)* Read *n*-bits from the register with $n \in \{64, 32, 16, 8\}$, and *2)* Write *n*-bits to the register with $n \in \{64, 32, 16, 8\}$.

Thus, we describe the possible change that occurs in one register with the set $S = \{w64, w32, w16, w8, 0\} \times \{r64, r32, r16, r8, 0\}$. 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., *r*64 implies *r*32). Note that we exclude 0, as it means the absence of any access.

SystemV ABI specifies 16 general purpose integer registers, thus for our purpose we represent the change occurring at the processor level as $\mathscr{S} = S^{16}$.

At last we declare a function, which calculates the change occurring in the processor state, when executing an instruction from $\mathscr{I}$: $decode : \mathscr{I} \mapsto \mathscr{S}$.

However, we do not go into detail how this function actually calculates this sate, because we rely on external libraries to perform this task. Implementing this function our self is out of scope due to the lengthy work required, as the x86-64 instruction set is quite large.

4.4 Calltarget Analysis

For either *count* or *type* policy to work, we need to arrive at an underestimation of the required parameters by any function existing within the targeted binary. We will employ a modified version of liveness analysis that tracks registers instead of variables to generate the needed underestimation. As our algorithm will be customizable, we look at the required merge functions to implement *count* and *type* policy. Furthermore, we need to eliminate the passing of variadic parameter lists from variadic functions, as this might cause our analysis to overestimate the required parameters.

Variable Liveness Analysis Theory. A variable is alive before the execution of an instruction, if at least one of the originating paths contains a read access before the variable is written to again. We employ liveness analysis, because we are looking for the parameters a function requires. This essentially requires read before write access, however global variables usually would also fall into this category, however these would not reside within parameter registers at the start of a function.

```
Function analyze(block : BasicBlock) : 𝒮^ℒ is
    state = Bl
    foreach inst ∈ block do
        state' = analyze_instr(inst)
        state = merge_v(state, state')
    end
    states = { }
    blocks = succ(block)
    foreach block' ∈ blocks do
        state' = analyze(block')
        states = states ∪ { state' }
    end
    state' = merge_h (states)
    return merge_v(state, state')
end
```

Figure 6: Algorithm used to analyze the liveness of a basic block.

Khedker et al. [**?**] defines live variable analysis on blocks, which we used to arrive the algorithm depicted in Figure 6 to compute the liveness state at the start of a basic block. We apply the liveness analysis for each function with the entry block of the function as start and the return blocks as end and after an analysis run for a function.

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

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

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

$$analyze\_instr : \mathscr{I} \mapsto \mathscr{S}^{\mathscr{L}} \tag{1c}$$

$$succ : \mathscr{I} \mapsto \mathscr{P}(\mathscr{I}) \tag{1d}$$

As the *analyze_instr* function calculates the effect of an instruction and is the heart of the analyze function. It will also

handle non jump and non fall-through successors, as these are not handled by DynInst in our case. We essentially have four cases that we handle: *1)* if the instruction is an indirect call or a direct call but we chose not follow calls, then return a state where all registers 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, *3)* if the instruction is a constant write (e.g., xor of two registers) then we remove the read portion before we return the decoded state, and *4)* 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 $\mathscr{S}^{\mathscr{L}}$ into parameters up to the following subsections.

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 following register state $S^{\mathscr{L}} = \{C, R, W\}$ which translates to the register super state $\mathscr{S}^{\mathscr{L}} = (S^{\mathscr{L}})^{16}$. We are only interested in the first occurrence of a R or W within one path, as following reads or writes do not give us more information. Therefore, our vertical merge function (*merge_v*) behaves in the following way that only when the first given state is C, 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 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.

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

This gives us the following register state $S^{\mathscr{L}} = \{C, r8, r16, r32, r64, W\}$ which translates to the register super state $\mathscr{S}^{\mathscr{L}} = (S^{\mathscr{L}})^{16}$.

As there could be more than one read of a register before it is written, we might be interested in more than just the first occurrence of a write or read on a path. To allow this we allow our merge operations to also return the value RW, which represents the existence of both read and write access and then can use W as an end marker of sorts. Therefore, our vertical merge operator conceptually intersects all read accesses along a path until the first write occurs (*merge_v*^{i}). In any other case it behaves like the previously mentioned vertical merge function.

Our horizontal merge(*merge_h*) function is again a simple pairwise combination of the given set of states, which are then combined with a union like operator with W preceding WR preceding R preceding C. Unless one side is W, read accesses are combined in such a way that always the higher one is chosen.

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

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

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

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

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

Ignoring Reads. When one instruction writes and reads a register at the same time we give the read access precedence, however there are exceptions (also mentioned in TypeArmor, however we expand slightly on that): *1)* `xor %rax, %rax` is the first obvious scenario, as it will always result in `%rax` holding the value 0, *2)* `sub %rax, %rax` is probably the next scenario, as it results also in `%rax` also holding the value 0,

and *3)* `sbb %rax, %rax` is also relevant, however it will not result in a constant value and based on the current state might either result in `%rax` holding the value 0 or 1.

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

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

Function *analyze(block : BasicBlock) :* $\mathscr{S}^{\mathscr{R}}$ **is**
    state = Bl
    foreach *inst* $\in$ *reversed(block)* **do**
        state' = analyze_instr(inst)
        state = merge_v(state, state')
    end
    states = { }
    blocks = pred(block)
    foreach *block'* $\in$ *blocks* **do**
        state' = analyze(block')
        states = states $\cup$ { state' }
    end
    state' = merge_h (states)
    return merge_v(state, state')
end

Figure 8: Algorithm used to analyze the reaching definitions of a basic block.

The book [**?**] defines reaching definition analysis on blocks, which we use to arrive at algorithm depicted in Figure 8 to compute the liveness state at the start of a basic block. We apply the reaching analysis at each indirect callsite directly before each call instruction.

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

$$merge\_v : \mathscr{S}^{\mathscr{R}} \times \mathscr{S}^{\mathscr{R}} \mapsto \mathscr{S}^{\mathscr{L}} \tag{2a}$$

$$merge\_h : \mathscr{P}(\mathscr{S}^{\mathscr{R}}) \mapsto \mathscr{S}^{\mathscr{R}} \tag{2b}$$

$$analyze\_instr : \mathscr{I} \mapsto \mathscr{S}^{\mathscr{R}} \tag{2c}$$

$$pred : \mathscr{I} \mapsto \mathscr{P}(\mathscr{I}) \tag{2d}$$

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

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

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

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

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

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

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

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

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

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

Our horizontal merge(*merge_h*) function is a simple pairwise combination of the given set of states, which are then

combined with a union like operator with T preceding S preceding U. When both states are set, we pick the higher one.

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

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

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

5 Implementation

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

6 Evaluation

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

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

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

6.1 Precision

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

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

6.1.1 Quality and Applicability of Ground Truth

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

O2	**calltargets**			**callsites**		
Target	match	Clang miss	tool miss	match	Clang miss	tool miss
proftpd	1015	0(0.0%)	15(1.45%)	155	0(0.0)	0(1.45)
vsftpd	318	0(0.0%)	0(0.0%)	14	0(0.0)	0(0.0)
lighttpd	290	0(0.0%)	311(51.74%)	66	0(0.0)	0(51.74)
nginx	921	0(0.0%)	0(0.0%)	266	0(0.0)	0(0.0)
mysqld	9742	13(0.13%)	3690(27.47%)	7923	24(0.3)	25(27.47)
postgres	6930	1(0.01%)	1512(17.91%)	687	1(0.14)	0(17.91)
memcached	133	0(0.0%)	91(40.62%)	48	1(2.04)	0(40.62)
node	20638	339(1.61%)	620(2.91%)	10965	29(0.26)	26(2.91)
geomean	1415.0	0.0 (0.0%)	0.0 (0.0%)	323.0	0.0 (0.0)	0.0 (0.0)

Table 1: Table shows the quality of structural matching provided by our automated verify and test environment, regarding callsites and calltargets when compiling with optimization level O2. The label Clang miss denotes elements not found in the data-set of the Clang/LLVM pass. The label tool miss denotes elements not found in the data-set of TYPESHIELD.

Call-targets. The obvious choice for structural comparison regarding calltargets is their name, as these are simply functions. First, we have to remove internal functions from our data-sets like the `_init` or `_fini` functions, which are of no consequence for us. Furthermore, while C functions can simply be matched by their name as they are unique through the binary, the same 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. We therefore need to reverse this process and then compare the fully typed names. Table 1 shows three data points regarding calltargets for the optimization level O2: *1)* The number of comparable calltargets that are found in both data sets, *2)* Clang miss: The number of calltargets that are found by TYPESHIELD but not by our Clang/LLVM pass, and *3)* tool miss: The number of calltargets that are found by our Clang/LLVM pass but not by TYPESHIELD

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

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

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

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

6.1.2 Classification Precision (*count*)

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

Experiment Setup (Call-targets). Union combination op-

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

Table 2: The results for analysis using the *count* policy on the O2 optimization level.

erator with an *analyze* function that follows into occurring direct calls. **Results (Call-targets).** 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 (Call-sites)** Union combination operator with an *analyze* function that does not follow into occurring direct calls while relying on a backward inter-procedural analysis. **Results (Call-sites).** The problem rate is under 0.01%, as there is only one test target, that exhibit a problematic classification. The rate of perfect classification is in general over 60% with Nginx (48.49%) and Node.js (56.34%) as an exception resulting in a geometric mean of 71.97%.

6.1.3 Classification Precision (*type*)

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

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

Table 3: The results for analysis using the *type* policy on the O2 optimization level.

Experiment Setup (Call-targets). 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 (Call-targets).** 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 (Call-sites).** Union combination operator with an *analyze* function that does not follow into occurring direct calls while relying on a backward inter-procedural analysis. **Results (Call-sites).** For two thirds of the set, the problem rate is under 2% and for last third it is not above 10%, resulting in a geomean of 1.38%. The rate of perfect classification is in general over 50% with Node.js (44.76%) as an exception resulting in a geometric mean of 68.35%.

6.2 Effectiveness

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

6.2.1 Theoretical Limits.

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

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

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

6.2.2 Reduction achieved by TYPESHIELD

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

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

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

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

Table 4: The results of comparing our implementation results with the theoretical limits for the different restriction policies combined with an address taken analysis for optimization level O2.

6.3 Runtime Overhead

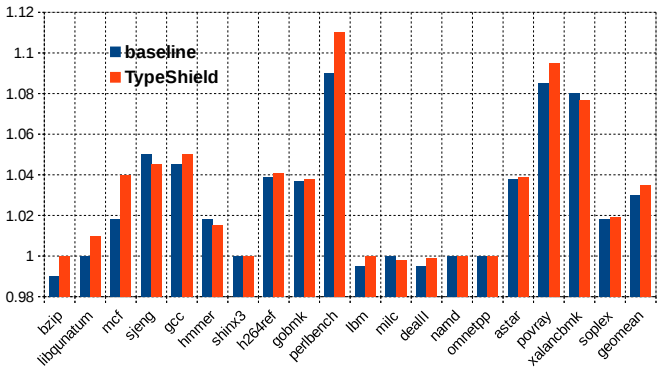

Figure 9: Benchmark run time normalized against the baseline for the SPEC CPU2006 benchmarks.

Figure 9 depicts the runtime normalized against the baseline for the SPEC CPU2006 benchmarks. In general, we have usually about 2%-5% performance drop when instrumenting using Dyninst. The reason for that are essentially cache misses introduced by jumping between the old and the new executable section of the binary generated by duplicating and patching the duplicate. This is necessary, because when out side of the compiler it is nigh on impossible to relocate indirect control flow, therefore every time an indirect control flow occurs, one jumps into the old executable section and from there back to the new executable section. Moreover, this is also dependent on the actual structure of the target, as it depends on the number of indirect control flow operations per time unit.

6.4 Instrumentation Overhead

The instrumentation overhead or the change in size due to patching is mostly due to the method Dyninst uses to patch binaries. Essentially the executable part of the binary is duplicated and extended with the patch. The usual ratio is around 40% to 60% while Postgres has an increase of 150% in binary size. One 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

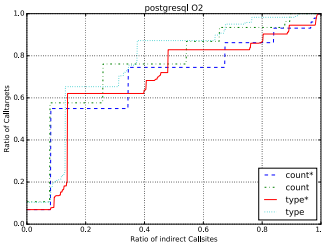

Figure 10: Postgresql -O2

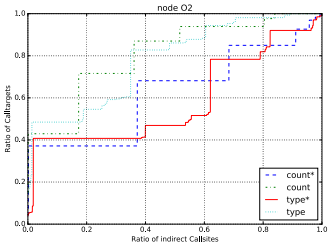

Figure 11: Node.js -O2

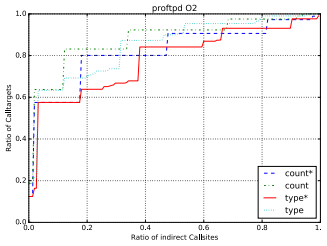

Figure 12: Proftpd -O2

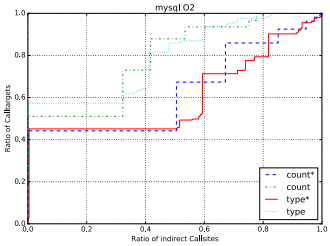

Figure 13: Mysql -O2

Figures, 10, 11, 12, and 13 depict the CDFs for the following programs: Postgresql, Node.js, Proftpd, and Mysql when compiled with the -O2 Clang compiler flag. We selected these four programs randomly. The CDFs depict the number of legal callsite targets and the difference between the type and the count policies. While the count policies have only a few number of changes, the number of changes that can be seen within the type policies is vastly higher. The reason for that is simple, the number of buckets that are used to classify the callsites and calltargets is simply higher. While type policies mostly perform better than the count policies, there are still parts within the type plot that are above the count plot, the reason for that is relatively simple: the maximum number of calltargets a callsite can access has been reduced, therefore a lower amount of calltargets is a higher percentage than be-

fore. However, all these results are dependent on the structure of the program.

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

6.6 Comparison with Other Tools

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

Table 5: The medians of calltargets per callsite for different tools.

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

7 Discussion

Comparison with TypeArmor. We are looking at two sets of results. First of all, we compare the overall precision of our implementation of the COUNT policy with the results from TypeArmor to set the perspective for the precision of our TYPE policy. We cannot compare data regarding overestimations of calltargets or underestimations of callsites, as TypeArmor did not provide sufficient data. The second point of comparison is the reduction of calltargets per callsite, however, this comparison is rather crude, as we most surely do not have the same measuring environment and not sufficient data to infer its quality.

Precision of Classification. TypeArmor reports a geometric mean of 83.26% for the perfect classification of calltargets regarding parameter count in optimization level O2, which compares rather well to our result of 82.24%. Regarding the perfect classification of callsites we report a geometric mean of 81.6% perfect classification regarding parameter count, while TypeArmor reports a geometric mean of 79.19%. However, we also have a geometric mean of about 7% regarding underestimations in the callsite classification with an upper bound of 16%, while TypeArmor reports that it does not incur underestimations in their callsites. Now, for our type based classification we incur the cost for two error sources. First, the error from the parameter count classification, which we base our type analysis on and second for the type analysis itself. The numbers for the perfect classification of calltargets regarding parameter types we report a 72.25% geometric mean of perfect classification, which is 87.85% of our precision regarding parameter counts. However, we report a geometric mean of 57.36% for perfect classification of callsites, which although seemingly low, is still 69.74% of our precision regarding parameter counts.

Reduction of Available Calltargets. While our count based precision focused implementation achieves a reduction in the same ballpark as TypeArmor regarding our test targets, lets us believe that our implementation of their classification schema is a sufficient approximation to compare against. However, we cannot safely compare those numbers, as the information regarding their test environment are rather sparse and the only data available is the median, which in our opinion does discard valuable information from the actual result set. This is the main reason we implemented an approximation, because we needed more metrics to compare TYPESHIELD and TypeArmor regarding calltargets. Using average and sigma, we can report that our precision focused type based classification can reduce the number of calltargets, by up to 20% more than parameter number based classification with an overall reduction of about 9%.

TypeArmor Discrepancies. As we have no access to source code of TypeArmor, we implemented an approximation of TypeArmor. Using this approximation we found some discrepancies between the data that we collected and data that was presented in the TypeArmor paper. A minor discrepancy between our results and the results of TypeArmor is that, while they basically implemented what we call a destructive merge operator for the liveness analysis. However, our data suggests that this operator is marginally inferior to the union path merge operator, when we compared them in our implementation. A major concern is the classification of calltargets, while we were able to reduce the number of overestimations of calltargets regarding parameter counts to essentially 0, the number of underestimations of calltarget did stay at a geometric mean of 7%. This error rate is rather large when compared to the reported 0% underestimation of TypeArmor, however we are not entirely sure what has caused this discrepancy. A possibility is the differing test environments, or a bug within our implementation that we are not aware of, or simply reaching definitions analysis alone is not the best possible algorithm for this particular problem.

Improving TYPESHIELD. To improve our type analysis, we see at least two possibilities. Incorporating refined data flow analysis and expanding the scope to also include memory. The main point of improvement is not the precision but for now more importantly the reduction of underestimations in the callsite analysis.

To refine the data flow analysis, we propose the actual

tracking of data values and simple operations, as these can be used to better differentiate the actual wideness stored within the current register. The highest gain, we see here would be the establishment of upper and lower bounds regarding values within the register, which would allow for more sophisticated callsite and calltarget invariants. Essentially we would have to resort to symbolic execution or some other sort of precise abstract interpretation.

Expanding the scope to also include memory, is another possible way of improving the type analysis, as it would allow us to distinguish normal 32-bit or 64-bit values and pointer addresses. Although we already have a limited approach of that in our reaching implementation, we still see room for improvement, as we only check whether a value is within one of three binary sections or 0.

Limitations of TYPESHIELD. First, we are limited by the capabilities of the DynInst instrumentation environment, the main problem, we are facing here is that non returning functions like exit are not detected reliably in some cases, which is why we were not able to test the 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 miss classifications for both calltargets and callsites.

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

8 Related Work

Type-Inference on Executables. Recovering variable types from executable programs is very hard in general for several reasons. First, the quality of the disassembly can very much from used framework to another. TYPESHIELD is based on DynInst and the quality of the executable disassembly fits our needs. For a more comprehensive review on the capabilities of DynInst and other tools we advice the reader to have a look at [**?**]. Second, alias analysis in binaries is undecidable in theory and intractable in practice [**?**]. There are several most promising tools such as: Rewards [**?**], BAP [**?**], SmartDec [**?**], and Divine [**?**]. These tools try with more or less success to recover type information from binary programs with different goals. Typical goals are: *i)* full program reconstruction (i.e., binary to code conversion, reversing, etc.), *ii)* checking for buffer overflows, *iii)* integer overflows and other types of memory corruptions. For a more exhaustive review of such tools we advice the reader to have a look at the review of Caballero et al. [**?**]. Interesting to notice is that the code from only a few of these tools is actually available.

While SmartDec seemed promising due to its simple type lattice that we wanted to leverage for our classification schema. Its integration into our DynInst based environment was not successful mostly for time constraints, as it was deemed to time consuming to extract the whole machinery and implement an interface to the DynInst disassembler. Therefore, we finally implemented our own version of type analysis and only focused on the wideness of the types, resulting in a simpler lattice than we initially wanted.

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

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

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

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

be able to enforce a precise and with low overhead primitive against COOP attacks.

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

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

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

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

Runtime based. "There is something available out there but I can not use it" *Anonymous*. Long story short conclusion: There are several promising runtime-based line of defenses against advanced CRAs but none of them can successfully protect against the COOP attack.

IntelCET [**?**] is based on, `ENDBRANCH`, a new CPU instruction which can be used to enforce an efficient shadow stack mechanism. The shadow stack can be used to check during program execution if caller/return pairs match. Since the COOP attack reuses whole functions as gadgets and does not violate the caller/return convention than the new feature provided by intel is useless in the face of this attack. Nevertheless, other highly notorious CRAs may not be possible after this feature will be implemented main stream in OSs and compilers.

Windows Control Flow Guard [**?**] is based on a user-space and kernel-space components which by working closely together can enforce an efficient fine-grained CFI policy based on a precomputed CFG. These new feature available in Windows 10 can considerably rise the bar for future attacks but in our opinion advanced CRAs such as COOP are still possible due the typical characteristics of COOP.

PathArmor [**?**] is yet another tool which is based on a precomputed CFG and on the LBR register which can give a string of 16 up to 32 pairs of from/to addressed of different types of indirect instructions such as `call`, `ret`, and `jump`. Because of the sporadic query of the LBR register (only during invocation of certain function calls) and because of the sheer amount of we think that against COOP this tool can not be used. First, because of the fact that the precomputed CFG does not contain edges for all possible indirect callsites which are accessed during runtime and second, the LBR buffer can be easily *tricked* by adding legitimate indirect callsites during the COOP attack.

Mitigation of Code-Reuse Attacks. In the last couple of years researchers have provided many versions of new Code Reuse Attacks (CRAs). These new attacks were possible since DEP [**?**] and ASLR [**?**] were successfully bypassed mostly based on Return Oriented Programming (ROP) [**?**, **?**, **?**] on one hand and on the other hand due to the discovery of new exploitable hardware and software primitives.

ROP started to present itself in the last couple of years in many faceted ways such as: Jump Oriented Programming (JOP) [**?**, **?**, **?**] which uses jumps in order to divert the control flow to the next gadget and Call Oriented Programming (COP) [**?**] which uses calls in order to chain gadgets together. CRAs have many manifestations and it is out of scope of this work to list them all.

On one hand, CRAs can be mitigated in general in the following ways: *i)* binary instrumentation, *ii)* source code recompilation and *iii)* runtime application monitoring. On the other hand, there is a plethora of tools and techniques which try to enforce CFI based primitives in executables, source code and during runtime. Next we briefly present the solution landscape together with the approaches and the techniques on which these are based: *a)* fine-grained CFI with hardware support, PathArmor [**?**], *b)* coarse-grained CFI used for binary instrumentation, CCFIR [**?**], *c)* coarse-grained CFI based on binary loader, CFCI [**?**] *d)* fine-grained code randomization, O-CFI [**?**], *e)* cryptography with hardware support, CCFI [**?**], *f)* ROP stack pivoting, PBlocker [**?**], *g)* canary based protection, DynaGuard [**?**], *h)* runtime and hardware support based on a combination of LBR, PMU and BTS registers CFIGuard [**?**], and *i)* source code recompilation with CFI and/or randomization enforcement against JIT-ROP attacks, MCFI [**?**], RockJIT [**?**] and PiCFI [**?**].

The above list is not exhaustive and new protection techniques can be obtained by combining available techniques or by using newly available hardware features or software exploits. However, none of the above mentioned techniques and tools can be used to mitigate COOP attacks.

9 Future Work

Structural matching capability. Improving the structural matching capability is in our opinion the most important further venue of research, as we need a reliable way to match a ground truth against the resulting binary. This is important because it is a prerequisite to the ability to generate reliable measurements and reduces the current uncertainty (i.e., we rely on the number of calltargets per callsite to match callsites and furthermore assume that the order within ground truth and binary is the same).

Better patching schema. Devising a patching schema that is based on Dyninst functionality, which allows annotation of calltargets so they can hold at least 4-bytes of arbitrary data. This is required to hold the type data that we generate using our classification. Keeping the runtime overhead of said patching schema low should be the second goal of this venue after satisfying stability.

Expanding to return values. Expanding our schema to return values is another viable venue of further work, as we were not able to reliably reduce the number of problematic classification regarding the return values of functions to manageable levels. Should one attempt this, it should be noted that the responsibilities of callsites and calltargets are reversed in this case: The callsite requires return value wideness, while the calltarget needs to provide it.

Using pointer/memory analysis. Introducing pointer/memory analysis to distinguish simple 32-bit and 64-bit values and actual addresses to even further restrict the possible number of calltargets per callsite. This would require more precise dataflow analysis, as in calculating value possibilities for registers at each instruction.

10 Conclusion

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