TypeShield: Practical Forward & Backward Edge Code Reuse Attack Defense

***Abstract*—Applications aiming for high performance and availability draw on several object oriented features available in the C/C++ programming language such dynamic object dispatch. However, there is an alarmingly high number of object dispatch corruption vulnerabilities which undercut security in significant ways and are in need of a thorough solution.**

In this paper we present, TypeShield, a binary runtime forward and backward edge protection tool which instruments program executables at load time. TypeShield enforces a novel runtime function parameter type and count control-flow integrity (CFI) policy in order to overcome the limitations of available approaches and to efficiently verify dynamic object dispatches and calltarget returns at runtime. To enhance practical applicability, TypeShield can be automatically and easily used in conjunction with legacy applications or where source code is missing to harden binaries. We evaluated TypeShield on database servers, FTP servers, memory caching applications and the SPEC CPU2006 benchmark and were able to efficiently and with low performance overhead protect these applications from forward and backward indirect edge corruptions. Finally, in a direct comparison with the state-of-the-art tools, TypeShield achieves higher caller/callee matching (*i.e.,* precision), while maintaining low runtime overhead and a calltarget set per callsite reduction gain of up to 35% compared to state-of-the-art.

I. Introduction

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

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

While the reasons for unwanted outcomes can be highly diverse, our work is primarily motivated by the presence of at least one exploitable memory corruption (*e.g.,* buffer overflow, etc.), which can enable the execution of sophisticated Code-Reuse Attacks (CRAs) such as the advanced COOP attack [2] and its extensions [3], [4], [5], [6]. A necessary ingredient for this class of attacks is the ability to corrupt the virtual pointer of an object in order to call gadgets by using a list of fake objects.

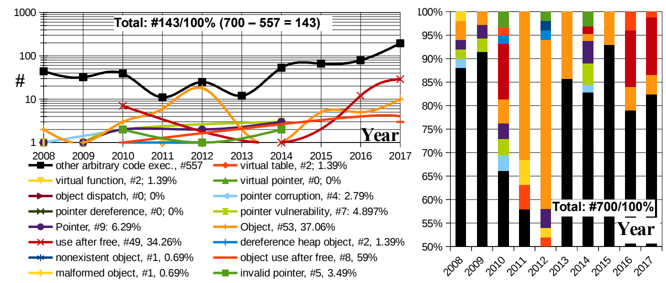

Fig. 1: Arbitrary Code Reuse Attacks vs. Object Corruptions.

To address such object dispatch corruptions[1] and in general any type of indirect control flow transfer violation, Control-Flow Integrity (CFI) [7], [8] was originally developed to secure indirect control flow transfers by adding runtime checks before each indirect callsite (*i.e.,* forward and backward). Unfortunately, COOP and its brethren bypass most deployed CFI-based enforcement policies, since these attacks do not exploit indirect backward edges (*i.e.,* return edges), but rather exploit the forward indirect control flow transfer imprecision which cannot be statically precisely determined upfront as alias analysis is undecidable [9] in program binaries.

More recent techniques and tools can be distinguished into those relying on *source code* access including SafeDispatch [10], ShrinkWrap [11], VTI [12], and IFCC/VTV [13]; the latter being used in production, but the reliance on source-code availability limits the applicability of the approach. In contrast, *binary*-based tools typically rely on forward-edge CFI policies. Examples include binCFI [14], [15], vfGuard [16], vTint [17], VCI [18], Marx [19] and TypeArmor [20].

TypeArmor, is based on a fine-grained forward edge CFI based policy relying on function parameter count checking during runtime. It calculates invariants for calltargets and indirect callsites based on the number of parameters provided at the callsite and consumed at the calltarget by leveraging static binary analysis. At the end of the analysis the binary is patched in order to enforce those invariants during

[1]Number (#, left side of Figure 1) and percentage (%, right side of Figure 1) of arbitrary code executions (ACE) reports related (all colors expect black) to pointer or virtual table (vptr/vtbl) corruption (see bag of words at the bottom of left Figure 1)* reported by U.S. NVD for the past 10 years [1]. In black are the ACE unrelated reports. X axis is years (left & right) and Y axis is number of reports in logarithmic scale (left, 143) and distribution in % of the reports (right, 700). As of May'17, U.S. NVD reports in total 700 ACEs from which 143 are the result of a vptr/vtable corruption (see * above) that are exploited by highjacking forward indirect calls. These vulnerabilities were reported in applications such as Google's Chrome & V8 JavaScript engine, Mozilla Firefox, Microsoft's IE 10, Edge & Chakra JavaScript engine, and iOS/MacOS apps.

runtime. While we believe that the general approach to be highly promising, we consider as a significant shortcoming that TypeArmor lacks precision with respect to the number of calltargets allowed per callsite which introduces significant inefficiencies (see §VII-F for more details). With our work, we aim to achieve both significant precision enhancements and calltarget set per callsite reduction.

In this paper, we present TYPESHIELD, a runtime binary-level fine-grained CFI tool for illegitimate forward calls, that is based on an improved forward-edge and backward-edge fine-grained CFI-policy compared to previous work [20]. TYPESHIELD does not rely on RTTI data (*i.e.,* metadata emitted by the compiler, most of the time stripped) or particular compiler flags, and is applicable to industrial software. TYPESHIELD takes the binary of a program as input and it automatically instruments it in order to detect illegitimate indirect calls at runtime. In order to achieve this, TYPESHIELD analyzes 64-bit binaries by focusing on function parameters which are passed with the help of registers. Based on the used ABI, TYPESHIELD is consequently able to track up to 6 function arguments for the Itanium C++ ABI [21] 64-bit calling convention. However, we stress that the presented methodology is usable for the ARM ABI [22] and Microsoft's C++ ABI [23] as well. Similarly to TypeArmor, we do not take into consideration floating-point arguments passed via xmm registers; which we want to address in future work. As we demonstrate in the evaluation section, this setup provides us with enough information to be significantly more precise than [20] when aiming to stop several state-of-the-art CRAs.

Analysis Description. More precisely, the analysis performed by TYPESHIELD: (1) uses for each function parameter its register wideness (*i.e.,* ABI dependent) in order to map calltargets per callsites, (2) uses an address taken (AT) analysis similar to [20] for all calltargets, and (3) compares individually parameters of callsites and calltargets in order to check if an indirect call transfer is acceptable or not, thereby providing a more fine-grained calltarget set per callsite compared to other state-of-the-art tools. TYPESHIELD uses automatically inferred parameter types which are used to construct a more precise construction of both the callee parameter types and callsite signatures. This is later used in the classification of matching callsites and calltargets. The result is a more precise callee target set for each caller than other state-of-the-art tools.

Analysis Details. The TYPESHIELD analysis is based on a use-def callees analysis to approximate the function prototypes, and liveness analysis at indirect callsites to approximate callsite signatures. This efficiently leads to a more precise control flow graph (CFG) of the binary program in question, which can be used also by other systems in order to gain a more precise CFG on which to enforce other types of CFI-related policies. These analysis results are used to determine a mapping between all callsites and legitimate calltarget sets. Further, this mapping is used in a backward analysis for determining the set of legitimate returns addresses for each function return determined by the each calltarget. Note that we consider each calltarget to be the start of function.

Forward Edge Policy. TYPESHIELD incorporates an improved protection policy which is based on the insight that if the binary adheres to the standard calling convention for indirect calls, undefined arguments at the callsite are not used by any callee by design. This further helps to reduce the possible target set of callees for each callsite.

Backward Edge Policy. TYPESHIELD uses a forward edge based propagation analysis used to determine a set of possible return addresses for function returns which follow the caller calle function calling convention. The policy is implemented as (1) a fast mode where a range is determined for each function return formed by the minimum and maximum value and as (2) slow mode where each return address of a function return is compared individually with the return address where the calltarget (i.e., function) wants to return. This represents a fine-grained alternative to SafeStack which was recently bypassed [24].

Comparison. TYPESHIELD uses partially different basic block analysis strategies than TypeArmor, and no control flow graph as TypeArmor does. Further, TYPESHIELD disallows an forward indirect call transfer where the types of the arguments provided are not super types of the arguments expected at the target. Also, it disallows backward edge indirect control flow transfers which do not point to addresses located after a callsite which is allowed to call the function (calltarget) containing the function return (backward edge starting point). Further, similar to TypeArmor, TYPESHIELD disallows forward indirect control flow transfers that prepares fewer arguments than the target callee consumes since otherwise it would risk breaking the binary. This invariants are used to enforce that each callsite targets only a strict calltarget set. Finally, the program binary hardened by TYPESHIELD contains a considerably reduced available calltarget set per callsite and return set per function returnsite, thus drastically limiting an attacker in his capabilities.

In summary, we make the following contributions:

- **Novel CFI-based protection technique.** We designed a novel fine-grained CFI technique for protecting forward and backward edges in a CFG against code reuse attacks.
- **Implemented an usable prototype.** We implemented, TYPESHIELD, a prototype which enforces the aforementioned technique in stripped program binaries. TYPESHIELD can serve as platform for developing other binary based protection mechanisms.
- **Evaluation.** We conduct a thorough set of evaluative experiments in which we show that TYPESHIELD is more precise and effective than other state-f-the-art tools. Further, we show that our tool has a higher calltarget set reduction per callsite, thus further reducing the attack surface.
- **Reproducibility of results.** We respond to calls emphasizing the importance of reproducibility of evaluation results (see NISTIR 7564 [25]) by releasing TYPESHIELD open source and by providing for each conducted experiment a precise description of the conducted experiment, thus increasing the reproducibility of our results.

II. BACKGROUND

In this section, we present a brief overview of the concept of C++-based polymorphism in §II-A and how indirect calls can be checked in practice in §II-C. In §V-A we present a forward edge function parameter count-based policy ([20]), and in §II-A2 we highlight security implications of indirect calls ,while in §VII-F we show that the state-of-the-art parameter count-based policy (§V-A) is imprecise w.r.t. to the enforced calltarget set per callsite. Finally, in §II-B we present in detail a real COOP attack.

A. Polymorphism in C++ Programs

Polymorphism, along inheritance and encapsulation, are the most used modern object-oriented concepts in C++. In C++, polymorphism allows accessing different types of objects through a common base class. A pointer of the type of the base object can be used to point to object(s) which are derived from the base class. In C++, there are several types of polymorphism: *a)* compile-time (or static, usually is implemented with templates), *b)* runtime (dynamic, is implemented with inheritance and virtual functions), *c)* ad-hoc (*e.g.,* if the range of actual types that can be used is finite and the combinations must be individually specified prior to use), and *d)* parametric (*e.g.,* if code is written without mention of any specific type and thus can be used transparently with any number of new types). The first two are implemented through early and late binding, respectively. In C++, overloading concepts fall under the category of *c)* and virtual functions, templates or parametric classes fall under the category of pure polymorphism. However, C++ provides polymorphism through: *i)* virtual functions, *ii)* function name overloading, and *iii)* operator overloading. In this paper, we are concerned with dynamic polymorphism, based on virtual functions (see ISO/IEC N3690 [26]), because it can be exploited to call: *x)* illegitimate virtual table entries (not) contained in the class hierarchy by varying or not the number of parameters and types, *y)* legitimate virtual table entries (not) contained in the class hierarchy by varying or not the number of parameters and types, and *z)* fake virtual tables entries not contained in the class hierarchy by varying or not the number of parameters and types. By legitimate and illegitimate virtual table entries we mean those virtual table entries which for a single indirect callsite lie in the virtual table hierarchy. More precisely, a virtual table entry is legitimate for a callsite if from the callsite to the virtual table containing the entry there is an inheritance path (see [11]). Virtual functions have several uses and issues associated, but for the scope of this paper we will look at the indirect callsites which are exploited by calling illegitimate virtual table entries (*i.e.,* functions) with varying number and type of parameters, *x)*. More precisely, *1) load-time enforcement:* as calling each indirect callsite (*i.e.,* callee) requires a fixed number of parameters which are passed each time the caller is calling, we enforce a fine-grained CFI policy by statically determining the number and types of all function parameter that belong to an indirect callsite, and *2) runtime verification:* as differentiating during runtime legitimate from illegitimate indirect caller/callee pairs requires parameter type (along parameter number), we check during run-time before each indirect callsite if the caller matches with the callee based on the previously added checks.

1) Exploiting Object Dispatches in C++: Figure 2 depicts a C++ code example (left) and how a COOP main-loop gadget (right) (*i.e.,* based either on ML-G (main-loop) or REC-G (recursive-gadget) or UNR-G (unrolled COOP gadget), see [3] for more details) is used to sequentially call COOP gadgets by iterating trough a loop controlled by the attacker.

```
1 class nsMultiplexInputStream final
2  :public nsIMultiplexInputStream //A0
3  ,public nsISeekableStream //A1
4  ,public nsIIPCSerializableInputStream //A2
5  ,public nsICloneableInputStream{ //A3
6 nsTArray<nsCOMPtr<nsIInputStream>> mStreams;
7 NS_IMETHODIMP nsMultiplexInputStream::Close(){
8   MutexAutoLock lock(mLock);
9   mStatus = NS_BASE_STREAM_CLOSED;
10  //set NS_OK flag
11  nsresult rv = NS_OK;
12  //get array length
13  uint32_t len = mStreams.Length();
14 //array-based main loop gadget (ML-G)
15 for (uint32_t i = 0; i<len; ++i){
16  //(0)hijacked object dispatch
17  nsresult rv2=mStreams[i]->Close();
18  if (NS_FAILED(rv2)) {
19      rv = rv2;
20  }
21 }
22  return rv;
23 }
```

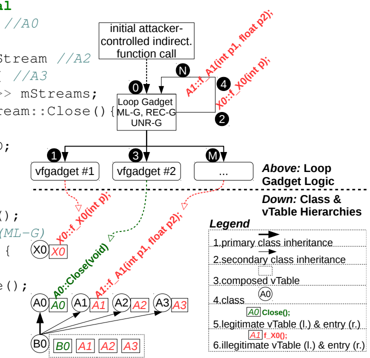

Fig. 2: COOP loop gadget (ML-G, REC-G, UNR-G) at work.

First, the object dispatch (see Figure 2 line 17) is exploited by the attacker in order to call different functions in the whole program by iterating on an array of fake objects previously inserted in the array.

Second, in order to achieve this the attacker previously exploits an existing program memory corruption (*e.g.,* buffer overflow) which is further used to corrupt an object dispatch, ❶, by inserting fake objects in the array and by changing the number of initial loop iterations. Next she invokes gadgets, ❶ and ❸ up to 🅜, through the calls, ❷ and ❹ up to 🅝, contained in the loop. As it can be observed in Figure 2 she can invoke from the same callsite legitimate functions (in total 🅝) residing in the virtual table (vTable) inheritance path (*i.e.,* at the time of writing this paper this type of information is particularly hard to recuperate from program binaries) 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.

Third, in this way different addresses contained in the program (1) (vTable) hierarchy (contains only virtual members), (2) class hierarchy (contains both virtual and non-virtual members) and (or) the whole program address space can be called. For example the attacker can call in any entry in the: (1) class hierarchy of the whole program, (2) class hierarchy containing only legitimate targets for this callsite, (3) virtual table hierarchy of the whole program, (4) virtual table hierarchy containing only legitimate targets for this callsite, (5) virtual table hierarchy and class hierarchy containing only legitimate targets for this callsite, and (6) virtual table hierarchy and class hierarchy of the whole program.

Finally, because there are no intrinsic language semantics—such as object cast checks—in the C++ programming language for object dispatches the loop gadget indicated in Figure 2 can be unconstrained used to call any possible entry in the whole program. Thus, making any program address a gadget part.

2) Security Implications of Indirect Calls: The C++ language standard (N3690 [26]) does not specify what happens when calling different virtual table entries from an indirect callsite. The standard says that we have a virtual function-related undefined behavior when: *a virtual function call uses an explicit class member access and the object expression refers to the complete object of x or one of that object's base class sub-objects but not x or one of its base class sub-objects*. As undefined behavior is not a clearly defined concept, we argue that in order to be able to deal with undefined behavior or unspecified behavior related to virtual function calls one needs to know how these language-dependent concepts are implemented inside the used compilers.

Forbidden forward-edge indirect calls are the result of a vPointer corruption. A vPointer corruption is not a vulnerability, but rather a capability which can be the result of a spatial or temporal memory corruption triggered by: (1) bad-casting [27] of C++ objects, (2) buffer overflow in a buffer adjacent to a C++ object or a use-after-free condition [2]. A vPointer corruption can be exploited in several ways. A manipulated vPointer can be exploited by pointing it in any existing or added program virtual table entry or into a fake virtual table which was added by an attacker. For example in case a vPointer was corrupted than the attacker could highjack the control flow of the program and start a COOP attack [2].

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

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

B. Real COOP Attack Example

The bug CVE-2014-3176 was exploited by Crane *et al.* [3] in order to perform a COOP attack, on the Google Chromium Web browser. The details of this attack are highly complex involving not properly handled interaction of browser extensions between the IPC, the sync API, and Google V8 engine and for this reason we briefly present a better documented COOP exploit which is in principle similar with this attack.

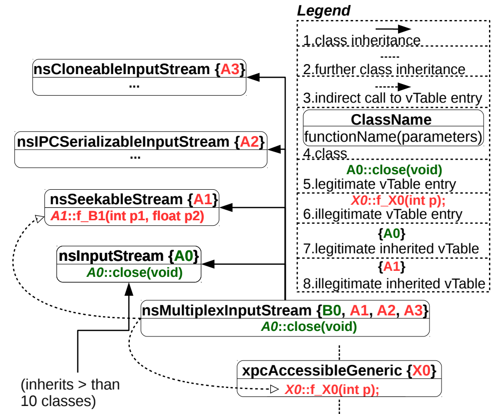

Fig. 3: Class hierarchy of classes used in the COOP attack.

Figure 3 depicts [2] a turing complete COOP attack [2] which was used to attack the Mozilla Firefox Web browser. By exploiting an existing buffer overflow bug the attacker was able to call into existing virtual table entries by having a main loop gadget at his disposal.

First, the attacker uses the C++ class `nsMultiplex-InputStream` (see Figure 3) which contains a main loop gadget (ML-G) inside the `nsMultiplexInput-Stream::Close(void)` function in order to perform indirect calls by dispatching calls on the fake objects contained in the array. The objects contained in the array during normal execution are of `nsInputStream` type and each of the objects will call the `Close(void)` function in order to close each of the previously opened streams.

Second, for performing the COOP attack, the attacker crafts a C++ program containing an array buffer holding six fake objects. These fake objects can call inside (and outside) the initial class and virtual table hierarchies with no constraints. During the attack a buffer is created in order to hold the fake objects. The crafted buffer will be used instead of the real code in order to call different functions available in the program code. For example, the attacker calls a function contained in the class `xpcAccessibleGeneric` which is not in the class hierarchy or virtual table hierarchy of the initially intended type of objects used inside the array. Moreover, the header file of this class (`xpcAccessibleGeneric`) is not included in the class `nsMultiplex-InputStream`.

Third, in total six fake objects are used to call into functions residing in unrelated class hierarchies with varying number of parameters and return types. The final goal of this attack is to prepare the program memory such that a Unix shell can be opened at the end of this attack.

Finally, this example illustrates why detecting vPointer corruptions is not trivial for real-world applications. As depicted in Figure 3, the class `nsInputStream` has 11 classes

[2]The class inheritance hierarchy of the classes involved in the COOP attack against the Firefox browser. Red letters indicate forbidden virtual table entries and green letters indicate allowed virtual table entries for the given indirect callsite contained in the main loop gadget.

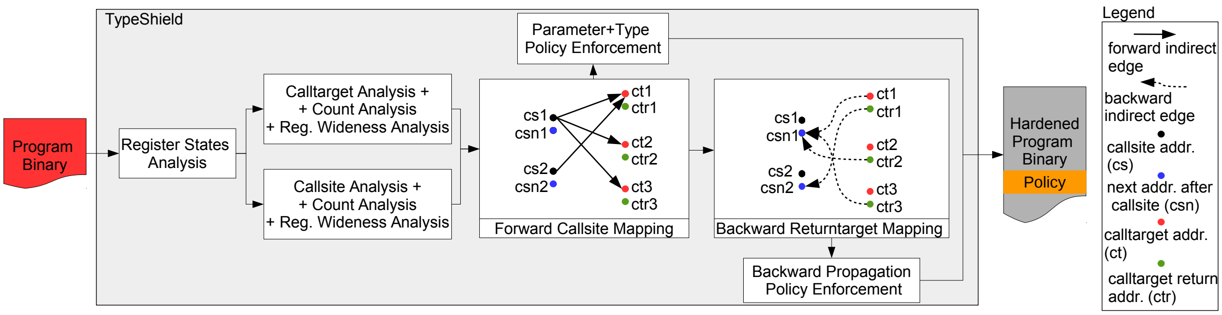

Fig. 4: System overview.

which inherit directly or indirectly from this class. The classes `nsSeekableStream`, `nsIPCSerializableInputStream` and `nsCloneableInputStream` provide additional inherited virtual tables which represent illegitimate calltargets for the initial `nsInputStream` objects and legitimate calltargets for the six fake objects which were added during the attack. Furthermore, declaration and usage of the objects can be widely spread out in the source code. This makes detection of the object types (*i.e.,* base class), range of virtual tables (*i.e.,* longest virtual table inheritance path for a particular callsite) and parameter types of the virtual table entries (*i.e.,* functions) in which it is allowed to call a trivial task for source code applications, but a hard task when one wants to apply similar security policies (*e.g.,* which rely on parameter types of virtual table entries) to binary executables.

C. Checking Indirect Calls in Practice

To the best of our knowledge, only the IFCC/VTV [13] tools (up to 8.7% performance overhead) is deployed in practice and can be used to check legitimate from illegitimate indirect forward-edge calls during runtime. vPointers are checked based on the class hierarchy. Furthermore, ShrinkWrap [11] (to the best of our knowledge not deployed in practice) is a tool which further reduces the legitimate virtual table ranges for a given indirect callsite through precise analysis of the program class hierarchy and virtual table hierarchy. Evaluation results show similar performance overhead but more precision with respect to legitimate virtual table entries per callsite. We noticed by analyzing the previous research results that the overhead incurred by these security checks can be very high due to the fact that for each callsite many range checks have to be performed during runtime. Therefore, in our opinion, despite its security benefit these types of checks cannot be applied to high performance applications.

A number of other highly promising tools (albeit also not deployed in practice) can overcome some of the drawbacks of the previously described tools. Bounov *et al.* [12] presented a tool ($\approx$ 1% runtime overhead) for indirect forward-edge callsite checking based on virtual table layout interleaving. The tool has better performance than VTV and better precision with respect to allowed virtual tables per indirect callsite. Its precision (selecting legitimate virtual tables for each callsite) compared to ShrinkWrap is lower since it does not consider virtual table inheritance paths. vTrust [28] (average runtime overhead 2.2%) enforces two layers of defense (virtual function type enforcement and virtual table pointer sanitization) against virtual table corruption, injection and reuse. TypeArmor [20] ($\leq$ than 3 % runtime overhead) enforces a CFI-policy based on runtime checking of caller/callee pairs and function parameter count matching. It is important to note that there are no C++ language semantics which can be used to enforce type and parameter count matching for indirect caller/callee pairs, this could be addressed with specifically intended language constructs in the future.

III. Threat Model

We align our threat model with the same basic assumptions as described in [20]. More precisely, we assume a resourceful attacker that has read and write access to the data sections of the attacked program binary. We also assume that the protected binary does not contain self-modifying code, handcrafted assembly or any kind of obfuscation. We also consider pages to be either writable or executable but not both at the same time. Further, we assume that the attacker has the ability to execute a memory corruption to hijack the program control flow. Finally, the analyzed program binary is not hand-crafted and the compiler which was used to generate the binary adheres to one of the standard calling conventions mentioned in the first section of this paper.

IV. Overview

In this section, we present the invariants for calltargets and callsites in §IV-A, and in §IV-B we present our function parameter type aware policy, while in §IV-C we highlight our backward edge protection policy. Finally, in §IV-D we talk about parameter count vs. parameter type policies.

A. Invariants

1) Calltargets and Callsites: We propose the following invariants for the function calltargets and callsites. (1) indirect callsites provide a number of parameters (*i.e.,* possibly overestimated compared to program source code), (2) calltargets require a minimum number of parameters (*i.e.,* possibly underestimated compared to program source code), and (3) the wideness of the callsite parameters has to be bigger or equal to the wideness of the parameters registers expected at the calltarget. In a nutshell the idea is that a callsite might only call functions that do not require more parameters than provided by the callsite and where the parameter register wideness of each parameter of the callsite is higher or equal to that parameter registers used at the calltarget. Figure 4 depicts this requirements by the forward indirect edges pointing from the black dots to the legitimate red dots.

2) Callertarget Returns: We propose the following invariant for the calltargets returns. (1) we enforce the caller caller convention between the calltarget return instruction and the address next to callsite which was used in first place to call that calltarget. Figure 4 depicts this analogy by the backward indirect edges pointing from the green dots to the legitimate blue dots.

B. Type Based Policy

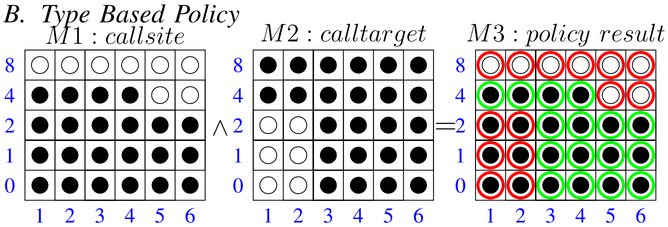

Fig. 5: Depicts our forward parameter type and count policy for a legitimate forward indirect control flow transfer. The X axis (parameter count) and Y axis (register wideness) of matrices $M1$, $M2$ and $M3$ represent function parameter count and bit-widths in bytes, respectively. Note that our type policy performs an $\wedge$ (*i.e.,* logical and) operation between each entry in $M1_{i,j}$ and $M2_{i,j}$ where i and j are column and row indexes. If two black filled circles located in $M1 \wedge M2$ overlap on positions $M1_i = M2_i \wedge M1_j = M2_j$ than we have a match. Green circles indicate a match whereas red circles indicate a mismatch in $M3$. Only if at least one match (green circle) is present in each of the matrix columns of $M3$ than the indirect call transfer will be allowed.

Figure 5 depicts the behavior of our type based policy when the callsite provides 6 parameters $\langle pcs1, ..., pcs6 \rangle$ having following bit wideness *pcs*1: 4-byte, *pcs*2: 4-bye, *pcs*3: 4-byte, *pcs*4: 8-byte, *pcs*5: 2-byte, *pcs*6: 2-byte, and the calltarget is expecting 6 parameters $pct1, ..., pct6$ having following bit wideness *pct*1: 4-byte, *pct*2: 4-bye, *pct*3: 0-byte, *pct*4: 0-byte, *pct*5: 0-byte, *pct*6: 0-byte of the expected parameters. TypeShield's type policy is defined as follows.

Definition 1. Let A be a calltarget ct_A and B a callsite cs_B than: $ct_A \subseteq cs_B \iff \forall\ i \subseteq [1, 6],\ wideness(parameter(\text{A})[i]) \leq\ wideness(parameter(\text{B})[i])$.

Whereas the policy of TypeArmor is the following.

Definition 2. Let A be a calltarget ct_A and B a callsite cs_B than: $ct_A \subseteq cs_B \iff \forall i \subseteq [1, 6],\ count(parameter(\text{A})) \leq count(parameter(\text{B}))$.

However, one can observe that the first policy (Definition 1) is more fine-grained than the second policy (Definition 2) since it performs checks for each parameter index in part separately whereas the second performs only a parameter count.

C. Backward Edge Policy

TypeShield uses a backward edge (*i.e.,* `retn`) fine-grained CFI protection policy which is based on two aspects. First, the forward edge addresses after each callsite are enforced to all legitimate calltargets return addresses (*i.e.,* function return address). Second, the caller callee calling convention which basically enforces that each function should return to the next address after the callsite that was used in first place to call that function is enforced. TypeShield uses two modes of operation for protecting the backward edge. The first mode (*i.e.,* fast path) is based on enforcing a range determined by the minimum and maximum of all legitimate (based on forward edge policy) return addresses. This range check is inserted before each function return of every calltarget (*i.e.,* function start address). Note that the fast path mode has a low runtime overhead but at the same time it could allow illegitimate return addresses due to the range check used. The second mode (*i.e.,* slow path) is based on enforcing a comparison check for each possible legitimate function return address determined by the forward edge policy. The slow path has a higher runtime overhead as the fast path since for each address a compare check is performed but it is more precise since illegitimate return addresses are excluded.

D. Parameter Count vs. Parameter Type

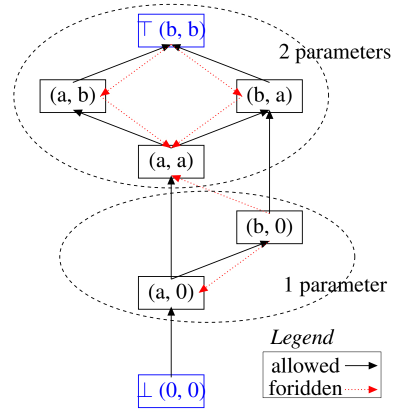

Fig. 6: Transition lattice between calltargets and callsites having two parameters. $a \wedge b \in \{0 - byte, 8 - byte, 16 - byte, 32 - byte, 64 - byte\}$ and the two function parameters have $\{0 - byte, 1 - byte, 2 - byte, 4 - byte, 8 - byte\}$ register wideness. TypeShield allows a transition from $a \rightarrow b\ iff$ $a_i \leq b_i$ where $i \in [1, 2]$. Note that $\top$ and $\bot$ represent the top and bottom elements of the lattice, respectively. An arrow represents an indirect control flow transfer from a callsite to a calltarget. The given lattice contains in total 8 black colored arrows (legal) and 6 red colored arrows (illegal) indirect control flow transitions. TypeShield allows only the legal transfers whereas [20] allows all of them.

Figure 6 depicts a subset of the total indirect control flow transfer space in any given C/C++ program represented as a lattice. In case a CFI policy schema is based on function parameter count with callsite overestimation and calltarget subestimation it is possible that a callsite can use any calltarget as long as the number of parameters provided and required are fulfilling the policy, even if the parameter types do not match (*i.e.,* consider a 8-bit value provided by the callsite but a 64-bit values required by the calltarget). Such a parameter count based policy is not precise [18] and would allow any call transfer inside the lattice space presented in Figure 6 and as such the calltarget set per callsite would be too permissive.

In order to effectively deal with this limitation we extend the above presented parameter count based policy to parameters types (*i.e.,* register wideness) as well. We introduce the following policy rules: (1) indirect callsites provide a maximum wideness to each parameter, and (2) calltargets require a minimum wideness for each parameter. Note that for both rules the minimum and maximum wideness for each function parameter is possibly underestimated compared to the source code of the program with which we also compare in §VII. Also note that the number of provided parameters must be not lower than the requirement number of consumed parameters. Finally, our approach is more fine-grained by considering parameter wideness and as such the allowed calltarget lattice space is considerably reduced since only the black arrows are allowed.

V. Design

In this section, we present in §V-A we present our function parameter count based policy, in §V-B, we present the details of our type policy, and in §V-C we introduce the definitions for our instructions analysis based on register states, in §V-D we present the design of our calltarget analysis, while in §V-E we depict the design of our callsite analysis. Finally, in §V-F we present our forward edge policy instrumentation strategy, and in §V-G we highlight our function backward edge analysis and policy instrumentation strategy.

A. Parameter Count Based Policy

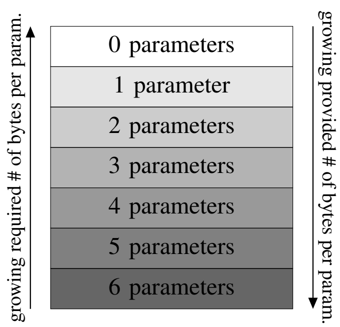

Fig. 7: Callsite & calltargets count policy classification schema.

Figure 7 depicts the used matching schema which shows that calltargets require parameters whereas callsites provide these parameters. Based on this schema we built a function parameter count-based policy which resembles the policy introduced by [20]. Calltargets are classified based on the number of parameters that these provide and callsites are classified by the number of parameters that these require.

Further, we consider the generation of high precision measurements for such classification with binaries as the only source of information rather difficult. Therefore, overestimations 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 Itanium C++ ABI [21]—designates as parameter registers. Furthermore, we do not consider floating point registers or multi-integer registers for simplicity reasons. The *count* policy is based on allowing 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. This is more precisely expressed as $\forall\, cs_1, cs_2;\, c_{cs_1} \leq c_{cs_2} \rightarrow \|\ \{ct \in \mathcal{F} \mid c_{ct} \leq c_{cs_1}\ \}\| \leq \|\ \{ct \in \mathcal{F} | c_{ct} \leq c_{cs_2}\}\|$.

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

B. Parameter Register Wideness Based Policy

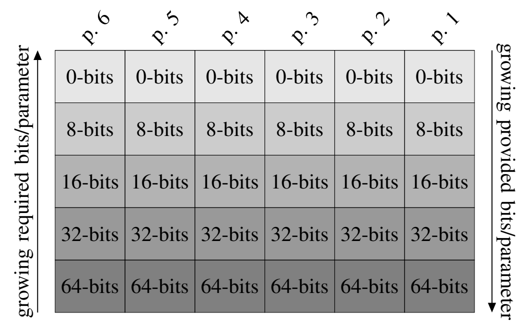

Fig. 8: Our type policy schema for callsites and calltargets. Note that p. means parameter. As it is depicted in this example, when requiring parameter width, one starts at the bottom of the above matrix and grows to the top, as it is always possible to accept more parameters than required. Also, the reverse is true for providing parameters, as it is possible to accept less parameters than provided. Note that accepting more parameters than provided is not allowed.

Figure 8 depicts the basic principle of our function parameter type policy. We use the register width as parameter type. As previously mentioned, there are 4 types of reading and writing accesses. Therefore, our set of possible types for parameters is $\texttt{TYPE} = \{64, 32, 16, 8, 0\}$; where 0 models the absence of a parameter. Since Itanium C++ ABI specifies 6 registers (*i.e.,* `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`) as parameter passing registers during function calls, we classify our callsites and calltargets into $\texttt{TYPE}^6$. Similar to our count policy, we allow overestimations of callsites and underestimations of calltargets, on the parameter types as well. Therefore, for a callsite cs it is possible to call a calltarget ct, only if for each parameter of ct the corresponding parameter of cs is not smaller w.r.t. the register width. This results in a finer-grained policy which is further restricting the possible set of calltargets for each callsite.

C. Analysis of Register States

Our register state analysis is register state based, another alternative would be to do symbol-based data-flow analysis which we will leave as future work. In order for the reader to understand our analysis we will first give some definitions. The set INSTR describes all possible instructions that can occur within the executable section of a program binary. In our case, this is based on the x86-64 instruction set. An instruction $i \in$ INSTR can non-exclusively perform two kinds of operations on any number of existing registers. Note that there are registers that can directly access the higher 8-bit of the lower 16-bit. For our purpose, we register this access as a 16-bit access. (1) Read n-bit from the register with $n \in \{64, 32, 16, 8\}$, and (2) Write n-bit to the register with $n \in \{64, 32, 16, 8\}$.

Next, we describe the possible change within one register as $\delta \in \Delta$ with $\Delta = \{w64, w32, w16, w8, 0\} \times \{r64, r32, r16, r8, 0\}$. Note that 0 represents the absence of either a write or read access and $(0, 0)$ represents the absence of both. Furthermore, wn or rn with $n \in \{64, 32, 16, 8\}$ implies all wm or rm with $m \in \{64, 32, 16, 8\}$ and $m < n$ (*e.g.,* $r64$ implies $r32$). Note that we exclude 0, as it means the absence of any access. Intanium C++ ABI specifies 16 general purpose integer registers. Therefore, we represent the change occurring at the processor level as $\delta_p \in \Delta^{16}$. In our analysis, we calculate this change for each instruction $i \in$ INSTR via the function $decode$: INSTR $\mapsto \Delta^{16}$.

D. Calltarget Analysis

Our calltarget analysis classifies calltargets according to the parameters they expect. Underestimations are allowed, however, overestimations are not permitted. For this purpose, we employ a customizable modified liveness analysis algorithm, which we will describe next. Furthermore, we will present certain corner cases we encountered, at the end of this section.

1) Liveness Analysis: A variable is alive before the execution of an instruction, if at least one of the originating paths performs a read access before any write access on that variable. If applied to a function, this calculates the variables that need to be alive at the beginning, as these are its parameters.

Algorithm 1: Basic block liveness analysis.

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

Function *analyze (block : INSTR*) :* $\mathcal{S}^{\mathcal{L}}$ **is**
    state = BI ▷ Initialize the state with first block
    foreach *inst* $\in$ *block* **do**
        state' = analyze_instr(*inst*) ▷ Calc. changes
        state = merge_h(*state*, *state'*) ▷ Merge changes
    end
    states = $\emptyset$ ▷ Set of succ. states
    blocks = successor(block) ▷ Get succ. blocks
    foreach *block'* $\in$ *blocks* **do**
        state' = analyze(*block'*) ▷ Analyze succ. block
        states = *states* $\cup$ {*state'*} ▷ Add succ. states
    end
    state' = merge_h (*states*) ▷ Merge succ. states
    return *merge_v*(*state*, *state'*) ▷ Merge to final state
end

Algorithm 1 is based on the liveness analysis algorithm presented in [29], which basically is a depth-first traversal of basic blocks. For customization, we rely on the implementation of several functions which we will present next. $\mathcal{S}^{\mathcal{L}}$ is the set of possible register states which depends on the specific implementations of the following operations.

- merge_v : $\mathcal{S}^{\mathcal{L}} \times \mathcal{S}^{\mathcal{L}} \mapsto \mathcal{S}^{\mathcal{L}}$, (merge vertically block states) describes how to merge the current state with the following state change.

- merge_h : $\mathcal{P}(\mathcal{S}^{\mathcal{L}}) \mapsto \mathcal{S}^{\mathcal{L}}$, (merge horizontally block states) describes how to merge a set of states resulting from several paths.

- analyze_instr : INSTR $\mapsto \mathcal{S}^{\mathcal{L}}$, (analyze instruction) calculates the state change that occurs due to the given instruction.

- succ : INSTR* $\mapsto \mathcal{P}$(INSTR*), (successor of a basic block) calculates the successors of the given block.

In our specific case, the function analyze_instr needs to also to handle non-jump and non-fall-through successors, as these are not handled by DynInst. Essentially, there are three relevant cases. First, if the current instruction is an indirect call or a direct call and the chosen implementation should not follow calls, then our analysis will return a state where all registers are considered to be written before read. Second, if the current instruction is a direct call and the chosen implementation should follow calls, then we start an analysis of the target function an return its result. If the instruction is a constant write (*e.g.,* xor of two registers), then we remove the read portion before we return the decoded state. Finally, in any other case, we simply return the decoded state. This leaves us with the two undefined merge functions and the undefined liveness state $\mathcal{S}^{\mathcal{L}}$.

2) Required Parameter Count: For our count policy, we need a coarse representation of the state of one register, thus we use the following representation. (1) W represents write before read access, (2) R represents read before write access, and (3) C represents the absence of access. Further, this gives us the $S^{\mathcal{L}} = \{C, R, W\}$ as register state, which translates to the register super state $\mathcal{S}^{\mathcal{L}} = (S^{\mathcal{L}})^{16}$. We implement merge_v in such a way that a state within a superstate is only updated if the corresponding register was not accessed, as represented by C. Our reasoning is that the first access is the relevant one in order to determine read before write. Our horizontal *merge(merge_h)* function is a simple pairwise combination of the given set of states, which are then combined with a union like operator with W preceding R and R preceding C. The index of highest parameter register based on the used call convention that has the state R considered to be the number of parameters a function at least requires to be prepared by a callsite.

3) Required Parameter Wideness: For our type policy, we need a finer representation of the state of one register as follows. (1) W represents write before read access, (2) $r8, r16, r32, r64$ represents read before write access with 8-, 16-, 32-, 64-bit width, and (3) C represents the absence of access. This gives us the following $S^{\mathcal{L}} = \{C, r8, r16, r32, r64, W\}$ register state which translates to the register super state $\mathcal{S}^{\mathcal{L}} = (S^{\mathcal{L}})^{16}$. As there could be more than one read of a register before it is written, we might be interested in more than just the first occurrence of a write or read on a path. To permit this, we allow our merge operations to also return the value RW, which represents the existence of both read and write access and then can use W with the functionality of an end marker. Therefore, our vertical merge operator conceptually intersects all read accesses along a path until the first write occurs $merge\_v^i$. In any other case, it behaves like the previously mentioned vertical merge function. Our horizontal merge $merge\_h$ function is a pairwise combination of the given set of states, which are then combined with a union-like operator with W preceding WR and WR preceding R and R preceding C. Unless one side is W, read accesses are combined in such a way that always the higher one is selected.

4) Encountered Analysis Issues:

a) Variadic Functions: Variadic functions are a special type of C/C++ functions 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`.

```
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>
```

Fig. 9: Assembly code of the `make_cmd` function which was compiled with Clang -O2 flag, and has a variadic parameter list which is shaded gray above.

Figure 9 depicts the binary code of a variadic function which allows an easier processing of parameters due to the fact that all potential variadic parameters are moved into a contiguous block of memory. Our analysis interprets this functions as a read access on all parameters and thus, we arrive at a potentially problematic overestimation. In our solution we opted to find these spurious reads and ignore them for now. A compiler will implement this type of operation very similar for all cases, thus we can achieve our desired outcome using the following steps: (1) we search for (what we call) the xmm-passthrough block, which entirely consists of moving values of registers `xmm0` to `xmm7` into contiguous memory, (2) we look at the predecessor of the xmm-passthrough block, which we call the entry block, next we check if the successors of the entry block consist of the xmm-passthrough block and the successor of the xmm-passthrough block (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.

b) 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 as follows: (1) `xor %rax, %rax` is the first scenario, as it will always result in `%rax` holding the value 0, (2) `sub %rax, %rax` is the second scenario, as it results in `%rax` also holding the value 0, and (3) `sbb %rax, %rax` is also relevant, however, it will not result in a constant value and based on the current state might either result in `%rax` containing 0 or 1.

E. Callsite Analysis

Our callsite analysis classifies callsites according to the parameters they provide. Overestimations are allowed, however, underestimations are not permitted. For this purpose we employ a customizable modified reaching definition algorithm, which we will show first. Furthermore, we will highlight some corner cases we encountered.

1) Reaching Definitions: An assignment to a variable is a reaching definition after the execution of a set of instruction if that variable still exists in at least one possible execution path. If applied to a callsite, this calculates the values that are provided by this callsite to the function it then invokes.

Algorithm 2: Basic block reaching definition analysis.

```
Input  : The basic block to be analyzed - block : INSTR*
Output: The reaching definition state - S^R

Function analyze(block : INSTR*) : S^R is
    state = BI                          ▷ Initialize the state with first block
    foreach inst ∈ reversed(block) do
        state' = analyze_instr(inst)    ▷ Calculate changes
        state = merge_v(state, state')  ▷ Merge changes
    end
    states = ∅                          ▷ Set of predecessor states
    blocks = pred(block)                ▷ Get predecessors blocks
    foreach block' ∈ blocks do
        state' = analyze(block')        ▷ Analyze pred. block
        states = states ∪ {state'}      ▷ Add pred. states
    end
    state' = merge_h(states)            ▷ Merge predecessors states
    return merge_v(state, state')       ▷ Merge to final state
end
```

Algorithm 2 is based on the reaching definition analysis presented in [29], which basically is a reverse depth-first traversal of basic blocks of a program. For customization, we rely on the implementation of several functions. $\mathcal{S}^{\mathcal{R}}$ is the set of possible register states which depends on the specific reaching definition implementation of the following operations.

- `merge_v` : $\mathcal{S}^{\mathcal{R}} \times \mathcal{S}^{\mathcal{R}} \mapsto \mathcal{S}^{\mathcal{R}}$, (merge vertically block states) describes how to merge the current state with the following state change.
- `merge_h` : $\mathcal{P}(\mathcal{S}^{\mathcal{R}}) \mapsto \mathcal{S}^{\mathcal{R}}$, (merge horizontally block states) describes how to merge a set of states resulting from several paths.
- `analyze_instr` : `INSTR` $\mapsto \mathcal{S}^{\mathcal{R}}$, (analyze instruction) calculates the state state change that occurs due to the given instruction.
- `pred` : `INSTR`* $\mapsto \mathcal{P}($`INSTR`$^*)$, (predecessor of a basic block) calculates the predecessors of the given block.

In our specific case, the function `analyze_instr` does not need to handle normal predecessors, as DynInst will resolve those for us. However there are several instructions that have to be handled as depicted in the following situations. (1) If the current instruction is an indirect call or a direct call but and the chosen implementation should not follow calls, then return a state where all registers are considered trashed. (2) If the instruction is a direct call and the chosen implementation should follow calls, then we start an analysis of the target function. (3) In all other cases we simply return the decoded state. This leaves us with the two merge functions and the undefined reaching definitions state $\mathcal{S}^{\mathcal{R}}$.

Previous work [29] provides a reaching definition analysis on blocks, which we use to arrive at the algorithm depicted in Algorithm 2 to compute the liveness state at the start of a basic block. We apply the reaching analysis at each indirect callsite directly before each call instruction.

This algorithm relies on various functions that can be used to configure its behavior. We define the function $merge\_v$, which describes how to compound the state change of the current instruction and the current state, the function $merge\_h$, which describes how to merge the states of several paths, the instruction analysis function $analyze\_instr$. Note, that the function $pred$, which retrieves all possible predecessors of a block is provided by the DynInst instrumentation framework.

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

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

2) Provided Parameter Count: For implementing our count policy, we use a coarse representation of the state of one register, thus we use the following representation. (1) T represents a trashed register, (2) S represents a set register (written to), and (3) U represents an untouched register. This gives us the following $S^{\mathcal{L}} = \{T, S, U\}$ register state which translates to the register super state $\mathcal{S}^{\mathcal{R}} = (S^{\mathcal{R}})^{16}$.

We are only interested in the first occurrence of a S or T within one path, as following reads or writes do not give us more information. Therefore, our vertical merge function $merge\_v$ behaves as follows. In case the first given state is U, than the return value is the second state and in all other cases it will return the first state.

Our horizontal merge $merge\_h$ function is a pairwise combination of the given set of states, which are then combined with a union like operator with T preceding S and 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 prepares at most.

3) Provided Parameter Width: In order to implement our type policy, we use a finer representation of the states of one register, thus we consider: (1) T represents a trashed register, (2) $s8, s16, s32, s64S$ represents a set register with 8-, 16-, 32-, 64-bit width, and (3) U represents an untouched register. This gives us the following $S^{\mathcal{L}} = \{T, s64, s32, s16, s8, U\}$ register state which translates to the register super state $\mathcal{S}^{\mathcal{R}} = (S^{\mathcal{R}})^{16}$.

However, 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 and S preceding U. Note, that when both states are set, we pick the higher one.

4) Encountered Analysis Issues: Our experiments with this implementation highlighted two issues. First, parameter lists with *holes* and address width underestimation. Second, register extension instructions can be also a cause for analysis problems. Finally, to reduce analysis runtime overhead, we also restricted the maximum path depth to 10 blocks.

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

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

F. Enforcing Our Forward Edge Policy

The result of the forward callsite and calltarget analysis is a mapping between the allowed calltargets for each callsite. In order to enforce this mapping during runtime each callsite and calltarget contained in the previous mapping are instrumented inside the binary program with two labels and a callsite located CFI-based checking mechanism. At each callsite the number of provided parameters are encoded as a series of six bits. At the calltarget the label contains six bits denoting how many parameters the calltarget expects. Additionally, at the callsite six bits encode which register wideness types each of the provided parameters have while at the calltarget another six bits are used to encode the types of the parameters expected. Further, at the callsite another bit is used to define if the function is expecting a `void` return type or not. All this information are written in labels before each callsite and calltarget. During runtime before each callsite these labels are compared by performing a xor operation between the bits contained in the previously mentioned labels. In case the xor operation returns false than the transfer is allowed else the program execution is terminated.

G. Backward Edge Analysis

In order to protect the backward edges of our previously determined calltargets for each callsite we designed an analysis which can determine possible legitimate return target addresses.

Algorithm 3: Calltarget return set analysis.

```
Input   : Forward edge callsite to calltargets map - fMap
Output: Backward edge to return addresses map - rMap
Function backwardAddressMapping(fMap) : rMap is
    ▷ visit all detected callsites in the binary
    foreach callsite ∈ fMap do
        ▷ get calltargets for callsite address key
        calltargetSet = getCalltargetSet(callsite, fMap)
        ▷ calltarget is the function start address
        ▷ visit all calltargets of a callsite
        foreach calltarget ∈ calltargetSet do
            ▷ get the next address after the callsite
            rTarget = getNextAddress(callsiteKey)
            ▷ find the address of function return
            rAddress = getReturnOfCalltarget(calltarget)
            ▷ rAddress is map key; rTarget is value
            rMap = rMap ∪ rMap add (rAddress, rTarget)
        end
    end
    ▷ return the backward edge addresses mappings
    return rMap
end
```

Algorithm 3 depicts how the forward mapping between callsites and calltargets is used to determine the backward address set for each return address contained in each address taken function. The $fMap$ is obtained after running the callsite and calltarget analysis (see §V-D and §V-E). These mapping contains for each callsite the legal calltargets where the forward edge indirect control flow transfer is allowed to jump to. This mapping is reflected back by construction a second mapping between the return address of each function for which we have the start address and a return target address set.

The return target address set is determined by incrementing the current address of the callsite for which the forward edge propagation algorithm is applied (*i.e.,* recall the caller callee calling convention). The $rMap$ is obtained by visiting each function return address and assigning to it the address next to the callsite which was used in order to transfer the control flow to the function in first place. At the end of the analysis all callsites and all function returns have been visited and a set for each function return address of backward edge addresses will be obtained. Note that the function boundary address (*i.e.,* retn) was detected by a linear search from the beginning of the function (calltarget) until the first return instruction was encountered. We are aware that other promising approaches for recuperating function boundaries (*e.g.,* [30]) exist, but we leave this as future work.

1) Enforcing The Backward Edge Policy: The previously determined $rMap$ in Algorithm 3 will be used to insert a check before each function return present in the $rMap$. We propose two modes of operation based on two types of checks which can be inserted before each function return instruction, depending on specific needs.

Fast path. Based on the $rMap$ for each function return the minimum and the maximum address out of the return set for a particular $rAdress$ return address will be determined. Next, these two values will be used to insert a range check having as left and right boundaries these two values. Before the return instruction of the function is executed the value of the function return is compared against these two values previously mentioned. In case the check fails than the program will be terminated else the indirect control flow transfer will be allowed. Note that this check has insignificant runtime overhead but on the other side it could contain not legitimate return addresses depending on the entropy of the $rAddress$es.

Slow path. Based on the $rMap$ for each function return a series of comparison checks are inserted in the binary. Before the return instruction of the function a series of comparison checks between the appropriate addresses stored in $rMap$ and the address where the function wants to return are performed. In case one of the check fails than the program will be terminated. The total number of comparison checks added is equal to the size of return address set which contains $rTarget$ values. Note that these types of checks are precise since only legitimate addresses are allowed but on the other side the runtime overhead is higher than in the case of the fast path because the number of checks is in general higher.

VI. Implementation

We implemented TypeShield using the DynInst (v.9.2.0) instrumentation framework. We currently restricted our analysis and instrumentation to x86-64 bit elf binaries using the Itanium C++ ABI call convention. We focused on the Itanium C++ ABI 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 support other ABIs as well. We developed the main part of our binary analysis pass in an instruction analyzer, which relies on the DynamoRIO [31] 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, which can be customized with relative ease, as we allow for arbitrary path merging functions. Next, we implemented a

Clang/LLVM (v.4.0.0, trunk 283889) back-end pass used for collecting ground truth data in order to measure the quality and performance of our tool. The ground truth data is then used to verify the output of our tool for several test targets. This is accomplished with the help of our Python-based evaluation and test environment. In total, we implemented TYPESHIELD in 5501 lines of code (LOC) of C++ code, our Clang/LLVM pass in 416 LOC of C++ code and our test environment in 3239 Python LOC.

VII. EVALUATION

We evaluated TYPESHIELD by instrumenting various open source applications and conducting a thorough analysis. Our test sample includes the two FTP server applications *Vsftpd* (v.1.1.0, C) and *Proftpd* (v.1.3.3, C), web server Lighttpd (v.1.4.28, C); FTP server Pure-ftpd (v.1.0.36, C) the two database server applications *Postgresql* (v.9.0.10, C) and *Mysql* (v.5.1.65, C++), the memory cache application *Memcached* (v.1.4.20, C), the *Node.js* application server (v.0.12.5, C++). We selected these applications in order to allow for comparison with [20]. In our evaluation we addressed the following research questions (RQs) w.r.t TYPESHIELD.

RQ1: How **precise** is it? (§VII-A)

RQ2: How **effective** is it? (§VII-B)

RQ3: What **runtime overhead** has does it have? (§VII-C)

RQ4: What is its **instrumentation overhead**? (§VII-D)

RQ5: What **security level** does it offer? (§VII-E)

RQ6: Which **upper bounds** can it enforce? (§VII-F)

RQ7: Is it **superior** compared to other tools? (§VII-G)

Comparison Method. We used TYPESHIELD to analyze each program binary individually. Next TYPESHIELD was used to harden each binary with forward and backward checks. The data generated during analysis and binary hardening was written external files for later processing. Finally, the previous obtained data was extracted with our Python based framework and inserted into spreedsheet files in order to better compare the results with other tools.

Experimental Setup. Our used setup consisted in a VirtualBox (version 5.0.26r) instance, in which we ran a Kubuntu 16.04 LTS (Linux Kernel version 4.4.0). We had access to 3GB of RAM and 4 out of 8 provided hardware threads (Intel i7-4170HQ @ 2.50 GHz).

A. Precision

In order to measure the precision of TYPESHIELD, we need to compare the classification of callsites and calltargets as provided by our tool with some ground truth data. We generated the ground truth data by compiling our test targets using a custom back-end Clang/LLVM compiler (v.4.0.0 trunk 283889) MachineFunction pass inside the x86-64-Bit code generation implementation of LLVM. During compilation, we essentially collect three data points for each callsite and calltarget as follows. (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.

1) Quality and Applicability of Ground Truth:

O2	calltargets			callsites		
Target	match	Clang miss	TypeShield miss	match	Clang miss	TypeShield miss
Proftpd	1202	0 (0%)	1 (0.08%)	157	0 (0)	0 (0.08)
Pure-ftpd	276	1 (0.36%)	0 (0%)	8	2 (20)	5 (0)
Vsftpd	419	0 (0%)	0 (0%)	14	0 (0)	0 (0)
Lighttpd	420	0 (0%)	0 (0%)	66	0 (0)	0 (0)
MySqld	9952	9 (0.09%)	7 (0.07%)	8002	477 (5.62)	52 (0.07)
Postgressql	7079	9 (0.12%)	0 (0%)	635	80 (11.18)	40 (0)
Memcached	248	0 (0%)	0 (0%)	48	0 (0)	0 (0)
Node.js	20337	926 (4.35%)	23 (0.11%)	10502	584 (5.26)	261 (0.11)
geomean	1460.87	4.07 (0.60%)	1.89 (0.40%)	203.77	9.04 (3.00)	6.37 (0.40)

TABLE I: 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 TypeShield denotes elements not found in the data-set of TYPESHIELD.

Table I depicts the results obtained w.r.t. the investigation of callrgets comparability and the callsites compatibility. We assessed the applicability of our collected ground truth, by assessing the structural compatibility of our two data sets.

Calltargets. The obvious choice for structural comparison regarding calltargets is their name, as these are functions. First, we have to remove internal functions from our datasets like the `_init` or `_fini` functions, which are of no relevance for this investigation. Furthermore, while C functions can simply be matched by their name as they are unique through the binary, the same cannot be said about the language C++. One of the key differences between C and C++ is function overloading, which allows defining several functions with the same name, as long as they differ in namespace or parameter type. As LLVM does not know about either concept, the Clang compiler needs to generate unique names. The method used for unique name generation is called mangling and composes the actual name of the function, its return type, its name-space and the types of its parameter list. Therefore, we need to reverse this process and then compare the fully typed names.

Table I shows three data points w.r.t. calltargets for the optimization level -O2: (1) Number of comparable calltargets that are found in both datasets, (2) Clang miss: Number of calltargets that are found by TYPESHIELD, but not by our Clang/LLVM pass, and (3) TypeShield miss: Number of calltargets that are found by our Clang/LLVM pass, but not by TYPESHIELD.

The problematic column is the Clang miss column, as these values might indicate problems with TYPESHIELD. These numbers are relatively low (below 1%) with only Node.js shows a noticeable higher value than the rest. The column labeled tool miss lists higher numbers, however, these are of no real concern to us, as our ground truth pass possibly collects more data: All source files used during the compilation of our test-targets are incorporated into our ground truth. The compilation might generate more than one binary and therefore, not necessary all source files are used for our test-target. Considering this, we can state that our structural matching between ground truth and TYPESHIELDs calltargets is very good.

Callsites. While our structural matching of calltargets is rather simple, matching callsites is more complex. Our tool can provide accurate addressing of callsites within the binary. However, Clang/LLVM does not have such capabilities in its intermediate representation (IR). Furthermore, the IR is not the final representation within the compiler, as the IR is transformed into a machine-based representation (MR), which is again optimized. Although, we can read information regarding parameters from the IR, it is not possible with the MR. Therefore, we extract that data directly after the conversion from IR to MR and read the data at the end of the compilation. To not unnecessarily pollute our dataset, we only considered calltargets, which have been found in both datasets.

Table I shows three data points regarding callsites for the optimization level -O2: (1) Number of comparable callsites that are found in both datasets, (2) Clang miss: Number of callsites that are discarded from the data set of TYPESHIELD, and (3) TypeShield miss: Number of callsites that are discarded from the data set of our Clang/LLVM pass. Both columns (Clang miss and TypeShield miss) show a relatively low number of encountered misses. Therefore, we can state that our structural matching between ground truth and TYPESHIELDs callsites is almost perfect.

2) Count Based Classification Precision:

O2 Target	Calltargets #	Calltargets perfect args	Calltargets perfect return	Callsites #	Callsites perfect args	Callsites perfect return
ProFtpd	1009	898 (88.99%)	845 (83.74%)	157	130 (82.8%)	113 (71.97%)
Pure-Ftpd	128	107 (83.59%)	52 (40.62%)	8	4 (50%)	8 (100%)
Vsftpd	315	270 (85.71%)	193 (61.26%)	14	14 (100%)	14 (100%)
Lighttpd	289	277 (95.84%)	258 (89.27%)	66	48 (72.72%)	57 (86.36%)
MySqld	9728	7138 (73.37%)	7845 (80.64%)	8002	5244 (65.53%)	6449 (80.59%)
Postgressql	6873	6378 (92.79%)	5241 (76.25%)	635	500 (78.74%)	573 (90.23%)
Memcached	133	123 (92.48%)	77 (57.89%)	48	47 (97.91%)	48 (100%)
Node.js	20069	16853 (83.97%)	14652 (73%)	10502	6001 (57.14%)	8841 (84.18%)
geomean	1097.06	952.62 (86.83%)	751.28 (68.48%)	203.77	150.16 (73.69%)	180.59 (88.62%)

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

Table II depicts 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.

Calltargets. For the first experiment we used a union combination operator with an $analyze$ function that follows into occurring direct calls. The results indicate a perfect classification of around 86% while for the returns it is over 68%.

Callsites. For the second experiment we used a union combination operator with an $analyze$ function that does not follow into occurring direct calls while relying on a backward inter-procedural analysis. The results indicate a rate of perfect classification of over 73% while for the returns it is over 88%.

3) Type Based Classification Precision:

Table III depicts 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.

O2 Target	Calltargets #	Calltargets perfect args	Calltargets perfect return	Callsites #	Callsites perfect args	Callsites perfect return
Proftpd	1009	835 (82.75%)	861 (85.33%)	157	125 (79.61%)	113 (71.97%)
Pure-Ftpd	128	101 (78.9%)	54 (42.18%)	8	4 (50%)	8 (100%)
Vsftpd	315	256 (81.26%)	179 (56.82%)	14	14 (100%)	14 (100%)
Lighttpd	289	253 (87.54%)	244 (84.42%)	66	48 (72.72%)	57 (86.36%)
MySqld	9728	6141 (63.12%)	7684 (78.98%)	8002	4477 (55.94%)	6449 (80.59%)
Postgressql	6873	5730 (83.36%)	4952 (72.05%)	635	455 (71.65%)	573 (90.23%)
Memcached	133	110 (82.7%)	70 (52.63%)	48	43 (89.58%)	48 (100%)
Node.js	20069	15161 (75.54%)	13911 (69.31%)	10502	4757 (45.29%)	8841 (84.18%)
geomean	1097.06	867.43 (79.06%)	723.70 (65.96%)	203.77	139.08 (68.25%)	180.59 (88.62%)

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

Calltargets. For the first experiment we used the union combination operator with an $analyze$ function that follow into occurring direct calls and a vertical merge and that intersects all reads until the first write. The results indicate a rate of perfect calltargets classification is over 79% while for the returns it is over 65%.

Callsites. For the second experiment we used the union combination operator with an $analyze$ function that does not follow into occurring direct calls while relying on a backward inter-procedural analysis. The results indicate a rate of perfect classification of over 68% while for the returns it is over 88%.

B. Effectiveness

We evaluated the effectiveness of TYPESHIELD by leveraging the results of several experiment runs. First, we established a baseline using the data collected from our Clang/LLVM pass. These are the theoretical limits of our implementation which can be reached for both the count and the type schema. Second, we evaluated the effectiveness of our count policy. Third, we evaluated the effectiveness of our type policy.

Table IV depicts the the average number of calltargets per callsite, the standard deviation σ and the median.

1) Theoretical Limits: We explore the theoretical limits regarding the effectiveness of the *count* and *type* policies by relying on the collected ground truth data, essentially assuming perfect classification.

Experiment Setup. Based on the type information collected by our Clang/LLVM pass, we conducted two experiment series. We derived the available number of calltargets for each callsite based on the collected ground truth applying the count and type schemes.

Results. (1) The theoretical limit of the *count** schema has a geometric mean of 129 possible calltargets, which is around 11% of the geometric mean of the total available calltargets (1097, see Table III), and (2) The theoretical limit of the *type** schema has a geometric mean of 105 possible calltargets, which is 9.5% of the geometric mean of the total available calltargets (1097, see Table III). When compared, the theoretical limit of the *type** policy allows about 19% less

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

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

available calltargets in the geomean with Clang -O2 than the limit of the *count** policy (*i.e.,* 105 vs. 129).

2) Calltarget Reduction per Callsite:

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

Results. (1) The *count* schema has a geometric mean of 166 possible calltargets, which is around 15% of the geometric mean of total available calltargets (1097, see Table III). This is around 28% more than the theoretical limit of available calltargets per callsite, see *count**, and (2) The *type* schema has a geometric mean of 144 possible calltargets, which is around 13% of the geometric mean of total available calltargets (1097, see Table III). This is around 37% more than the theoretical limit of available calltargets per callsite, see *type**. Our implementation of the *type* policy allows around 21% less available calltargets in the geomean with Clang -O2 than our implementation of the *count* policy and further a total reduction of more than 87% (141 vs. 1097) w.r.t. to total number of AT calltargets available after our *count* and *type* policies were applied.

C. Runtime Overhead

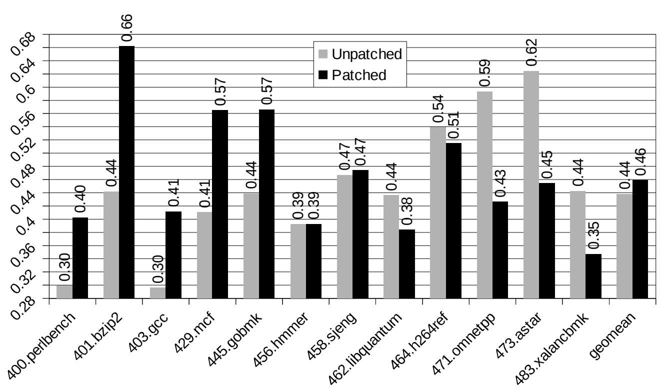

Fig. 10: SPEC CPU2006 Benchmark Results.

Figure 10 depicts the runtime overhead obtained by applying our tool to several SPEC CPU2006 benchmarks in count mode and in count and type mode, respectively. The obtained geomean runtime overhead is around 4% performance overhead when instrumenting using DynInst. One reason for the performance drop includes cache misses introduced by jumping between the old and the new executable section of the binary generated by duplicating and patching. This is necessary, because when outside of the compiler, it is nearly impossible to relocate indirect control flow. Therefore, every time an indirect control flow occurs, one jumps into the old executable section and from there back to the new executable section. Moreover, this is also dependent on the actual structure of the target, as it depends on the number of indirect control flow operations per time unit. Another reason for the slightly higher (yer acceptable) performance overhead is due to our runtime policy which is more complex than that of other state-of-the-art tools. However, the runtime overhead of TypeShield (4%) is comparable with other state-of-the-art virtual table defense tools such as: TypeArmor (3%), VCI [18] (7.79% overall and 10.49% on only the SPEC CPU2006 programs), vfGuard [16] (10% - 18.7%), T-VIP [32] (0.6% - 103%), SafeDispatch [10] (2% - 30%), and VTV/IFCC [13] (8% - 19.2%). Finally, this results qualify TypeShield as a highly practical tool.

D. Instrumentation Overhead

The instrumentation overhead (*i.e.,* binary blow-up) 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 check we insert. The usual ratio we encountered in our experiments is around 40% to 60% with Postgres having an increase of 150% in binary size. One cannot reduce that value significantly, because of the nature of code relocation after loosing the information which 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. Finally, this results should not represent an issue for memory resourceful systems on which these applications typically run.

E. Security Analysis

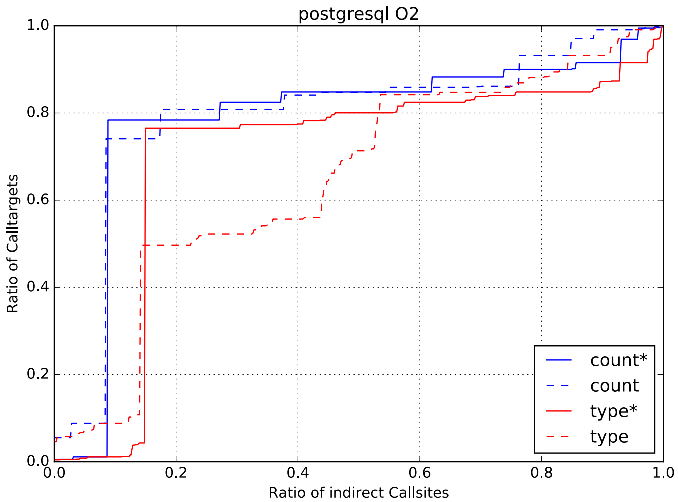

Fig. 11: CDF for Postgresql compiled with Clang -O2.

Figures 11 depicts the CDF of the Postgresql program which was compiled with the Clang -O2 flag. We selected this program randomly from our sample programs. The CDFs depict the number of legal callsite targets and the difference between the type and the count policies. While the count policies have only a few changes, the number of changes that can be seen within the type policies are vastly higher. The reason for this is fairly straightforward: the number of buckets that are used to classify the callsites and calltargets is simply higher. While type policies mostly perform better than the count policies, there are still parts within the type plot that are above the count plot, the reason for that is also relatively simple: the maximum number of calltargets a callsite can access has been reduced. Therefore, a lower number of calltargets is a higher percentage than before. However, Figure 11 depicts clearly that the *count** and *type** have higher values as *count* and *type*, respectively. This further, confirms our assumptions w.r.t. these used metrics. Finally, note that the results dependent on the particular internal structure of the hardened programs.

F. TyperShield Upper Bounds

In this section we briefly relate the upper bounds values of TYPESHIELD with the ones of TypeArmor. TypeArmor [20] enforces a CFI-based runtime policy in a binary for constraining object dispatches at the callsite based on function parameter count checks. We believe that their callsite vs. calltarget set enforcing policy is too permissive and thus many illegitimate indirect forward edge based control flow transfers are possible.

Let us consider the following example, where each callsite is preparing, say, $p = 6 \in [1, 6]$ parameters. Then TypeArmor policy would allow calltargets which consume the same number of parameters as prepared, $c = 6 \in [1, 6]$ or a lower value. Thus all possible numerical parameter mismatches are allowed by TypeArmors policy as long as p is greater or equal than c.

- TypeArmor ***ideally*** would allow for a single callsite a set of calltargets containing a maximum of 4096 possibilities if we consider the maximum value of provided parameters to be $p = 6$ (due to $p \in [1, 6]$ possible provided parameters). Now, consider 4 C++ integer parameter types t which use: 8, 16, 32 and 64 byte function parameter register wideness. Thus, we obtain $t^p = 4^6 = 4096$ allowed calltargets per callsite if TypeArmor is used. Note that for simplicity reasons we considered $t = 4$ but in practice t is often even larger since there are many types of parameters in C++. Thus, all these data types are ignored by TypeArmor.
- TypeArmor ***actually*** allows more than t^p calltargets per callsite. If we have $t = 4$ integer types due to TypeArmors overestimation and underestimation we get for each callsite an additional number of calltargets. Let $p = 6$, then we get $c = 6x+5y+4z+3t+2p+1v$ where: x is the sum of all calltargets consuming 6 parameters, y is the sum of all calltargets consuming 5 parameters and so on down to 0 parameters. Note that this holds since TypeArmor allows more parameters to be provided than consumed by the calltarget. Then, $c = 2100 = 600+500+400+300+200+100 \; iff \; x = y = z = t = p = v = 100$. Note that $x = 100$ is feasible number under realistic conditions in large applications (*i.e.,* Google Chrome, Firefox). Next 2100 is added to 4^6. Thus, for a single callsite providing $p = 6$ parameters TypeArmor allows theoretically in total $4^6 + 2100 = 6196$ calltargets for each callsite. Similar reasoning applies to $p = 5$ where we get $4^5+(1500 = 500+400+300+200+100) = 1024+1500 = 2524 \; iff \; x = y = z = t = p = v = 100$ allowed calltarget per callsite, or $p \in [1, 4]$, too.

Finally, as it can be observed TypeArmor is too permissive (see also Figure H.1 [18] for more details), thus we present TYPESHIELD, a more precise alternative. TYPESHIELD can deal with the aforementioned 4 types of register widths and can further reduce the the legitimate calltarget set per callsite as shown herein.

G. Comparison with Other Tools

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

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

TABLE V: Medians of calltargets per callsite for different tools. Note that the smaller the geomean numbers are, the better the technique is. AT is a technique which allows calltargets that are address taken. IFCC is a compiler based solution and depicted here as a reference for what is possible when source code is available. TypeArmor and TypeShield on the other hand are binary-based tools. We can observe that our type-based tool reduces the number of calltargets by up to 35% when compared to the AT method and by 13% on average when comparing with TypeArmor.

Nevertheless, TYPESHIELD represents a strong improvement w.r.t. calltarget per callsite reduction in binary programs.

VIII. DISCUSSION

A. Comparison with TypeArmor

1) 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 35% more than parameter number based classification with an overall reduction of about 13%.

B. TypeArmor Discrepancies

Next we will report some discrepancies which we think are worthwhile to be mentioned herein. A minor discrepancy between our results and the results of TypeArmor is that, while TypeArmor basically implements 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 around 5%. 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 scenario.

C. Improving TYPESHIELD

First, incorporating a refined data flow analysis and expanding the scope of the analysis include memory analysis as well. The main point of improvement is not the precision but for now more importantly the reduction of underestimations in the callsite analysis. In order 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.

Second, expanding the scope to include memory analysis, is another possible way of improving the type analysis, as it would allow 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. Further, we see room for improvement, as we only check whether a value is within one of three binary sections or 0.

D. Limitations of TYPESHIELD

First, TYPESHIELD is limited by the capabilities of the DynInst instrumentation environment, where non-returning functions like exit are not detected reliably in some cases. As a result, we cannot test the Pure-FTP server, as it heavily relies on these functions. The problem is that those non-returning functions usually appear as a second branch within a function that occurs after the normal control flow, causing basic blocks from the following function to be attributed to the current function. This results in a malformed control flow graph and erroneous attribution of callsites and problematic mis-classifications for both calltargets and callsites.

Second, TYPESHIELD draws on variety within the binary. In particular, we rely on the fact that functions use more than only 64-bit values or pointers within their parameter list, otherwise, TYPESHIELD is equivalent to a parameter count-based implementation. Occurrences of such situations are quite rare, as we learned with our experiments. With a study based on source level information, we could not detect a difference between our *type* policy and a *count* policy. However, when using our tool, we were able to detect differences, which reinforce the fact, that we do not rely on declaration of parameters, but usage of those.

Third, TYPESHIELD can protect forward indirect edges in a binary program and can complement a shadow stack [33] protection technique. For this reason, we assume that TYPESHIELD can run side by side with an ideal backward-edge protection mechanism such as a shadow stack [34]. However, the main goal of TYPESHIELD is to complement shadow stack based defenses which fail to account for attacks not violating the backward-edge calling conventions such as the COOP attack.

Fourth, TYPESHIELD is not intended to be more precise than source code based tools such as IFCC/VTV [13].

However, TYPESHIELD is highly useful in situations where the source code is typically not available (*e.g.,* off-the-shelf programs), where programs rely on third party libraries, and where the recompilation of all the shared libraries is not possible. Further, binary based tools such as TYPESHIELD can offer precise protection when source code is not available or recompilation is not feasible or desirable.

Finally, while a major step forward, TYPESHIELD cannot thwart all possible attacks, as even solutions with access to source code are unable to protect against all possible attacks [35]. In contrast, TYPESHIELD, our binary-based tool, can stop all currently COOP attacks published to date and significantly raises the bar for an adversary when compared to TypeArmor and similar tools. Moreover, TYPESHIELD provides a strong mitigation for other types of code-reuse attacks as well which violate the caller callee calling convention.

IX. RELATED WORK

A. 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 [36]. Second, alias analysis in binaries is undecidable in theory and intractable in practice [37]. There are several most promising tools such as: Rewards [38], BAP [39], SmartDec [40], and Divine [41]. 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, and (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.* [42]. Interesting to notice is that the code from only a few of these tools is actually available.

SmartDec seemed at first 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 several reasons, 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 function parameter type analysis and focused only on the wideness of the types, resulting in a simpler lattice than we initially intended.

B. Mitigation of Advanced CRAs

Mitigation of Forward-Edge based Attacks. Recursive-COOP [3], COOP [2], Subversive-C [4] and the attack of Lan *et al.* [6] are forward-edge based CRAs which cannot be addressed with: (i) shadow stacks techniques and hardware-based approaches such as Intel CET [43] (*i.e.,* since advanced COOP do not violate the caller/callee convention), (ii) coarse-grained Control-Flow Integrity (CFI) [7], [8] techniques, and (iii) OS-based approaches such as Windows Control Flow Guard [44] since the precomputed CFG does not contain edges for indirect callsites which are explicitly exploited during the COOP attack.

1) Binary Based Techniques: The following tools address vTable protection through binary instrumentation, but fail to mitigate against COOP: vfGuard [16], and vTint [17]. The only binary-based tool which we are aware of that can protect against COOP is TypeArmor [20]. TypeArmor uses a fine-grained CFI policy based on caller/callee (but only indirect callsites) matching, which checks during runtime if the number of provided and needed parameters match.

TYPESHIELD is related to TypeArmor [20], since we also enforce strong binary-level invariants on the number of function parameters. Further, TYPESHIELD also aims for exclusive protection against advanced exploitation techniques, which can bypass fine-grained CFI schemes and vTable protections at the binary level. However, TYPESHIELD offers a better restriction of calltargets to callsites, since we not only restrict based on the number of parameters, but also on the width of their types. This results in much smaller buckets that in turn can only target a smaller subset of all address-taken functions.

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.

VCI [18] is a binary based tool (7.9%) based on DynInst which can protect forward edge indirect control flow violations based on reconstructing a quasi program class hierarchy (*i.e.,* no class root node and the edges are not directed). The authors claim that VCI is 10 times more precise w.r.t. reducing the calltarget set per callsite. In contrast to TYPESHIELD VCI can not protect backward edge violations and we arguably due to the conservative analysis the VCI could skip some corner situations allowing not legitimate calltargets.

Marx [19] is most similar to VCI and as VCI this tool reconstructs the same type of quasi program class hierarchy. No runtime efficiency numbers were provided in the paper. The authors claim that Marx can recuperate a class hierarchy which is more precise than that of IDAPro. The paper is geared towards first providing a tool which can be used by analyst in order to reverse engineer a binary. The precision of the calltarget set reduction per callsite should be similar to those of VCI but no comparison was compared in the paper. Compared to TYPESHIELD Marx can not against backward edge violations and arguably suffers from the same limitations as VCI.

VTPin [45] is a runtime based tool ($\approx$5%) used for protecting against VTable hijacking, via use-after-free vulnerabilities. VTPin pins all the freed VTable pointers on a safe VTable under VTPin's control. For each object deallocation, VTPin deallocates all space allocated, but preserves and updates the VTable pointer with the address of the safe VTable. As consequence a dangling pointer can invoke invoke a method provided by VTPin's safe object. TPin needs to keep track of meta-data in order to detect runtime dangling pointer violations. The tool can not protect against the COOP attack since the COOP attack does not rely on dangling pointers. In contrast with TypeShield this tool can not protect against backward edges violations.

In this paper, rather than claiming that the invariants offered by TypeShield are sufficient to mitigate all versions of the COOP (as [20] does) attack we conservatively claim that TypeShield further raises the bar w.r.t. what is possible when defending against COOP attacks on the binary level.

2) Source Code Based Techniques: Indirect callsite targets are checked based on vTable integrity. Different types of CFI policies are used such as in the following tools: SafeDispatch [10], IFCC/VTV [13] LLVM and GCC compiler. Additionally, the Redactor++ [3] uses randomization vTrust [28] checks calltarget function signatures, CPI [46] 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 [11], IFCC/VTV [13], SafeDispatch [10], vTrust [28], Readactor++ [3], CPI [46] and the tool presented by VTI [12]. These tools profit from high precision since they have access to the full semantic context of the program though the scope of the compiler on which they are based. Because of this reason, these tools target mostly other types of security problems than binary-based tools address. For example, some of the last advancements in compiler based protection against code reuse attacks address mainly performance issues. Currently, most of the above presented tools are only forward edge enforcers of fine-grained CFI policies with an overhead from 1% up to 15%.

3) Runtime Based Techniques: Several promising runtime-based defenses against advanced CRAs exist but currently none of them can successfully protect against the COOP attack.

IntelCET [43] 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 [44] 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 [47] 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 data which passes through the LBR register this approach has in our opinion a fair potential to catch different types of CRAs but we think that against COOP this tool can be used only with limited success. First, because of the fact that the precomputed CFG does not contain edges for all possible indirect callsites which are accessed during runtime. Second, the LBR buffer can be easily fooled by interleaving legitimate with illegitimate indirect callsites during the COOP attack.

C. Mitigation of not Advanced CRAs

In the last couple of years researchers have provided many versions of new Code Reuse Attacks (CRAs). These new attacks were possible since DEP [48] and ASLR [49] were successfully bypassed mostly based on Return Oriented Programming (ROP) [50], [51], [52] on one hand and due to the discovery of new exploitable hardware and software primitives, on the other hand.

ROP started to present itself in the last couple of years in many faceted ways such as: Jump Oriented Programming (JOP) [53], [54], [55] which uses jumps in order to divert the control flow to the next gadget and Call Oriented Programming (COP) [56] 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.

First, CRAs can be mitigated in general in the following ways: (i) binary instrumentation, (ii) source code recompilation and (iii) runtime application monitoring. Second, there is a plethora of tools and techniques which try to enforce CFI based primitives in executables, source code and during runtime. Thus, 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 [47], (b) coarse-grained CFI used for binary instrumentation, CCFIR [14], (c) coarse-grained CFI based on binary loader, CFCI [57] (d) fine-grained code randomization, O-CFI [58], (e) cryptography with hardware support, CCFI [59], (f) ROP stack pivoting, PBlocker [60], (g) canary based protection, DynaGuard [61], (h) runtime and hardware support based on a combination of LBR, PMU and BTS registers CFIGuard [62], and (i) source code recompilation with CFI and/or randomization enforcement against JIT-ROP attacks, MCFI [63], RockJIT [64] and PiCFI [65].

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, notice that none of the above mentioned techniques and tools can be used to mitigate COOP attacks.

X. Future Work

1) Improving the Structural Matching: 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).

2) Improving the Patching Schema: Devising a patching schema that is based on Dyninst functionality, which allows annotation of calltargets so they can hold at least 4-byte 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.

3) Using Pointer and 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 data flow analysis, as in calculating value possibilities for registers at each instruction.

4) Filtering Forward Edges: As depicted in [18] and [19] it is possible to reconstruct a quasi class hierarchy (*i.e.,* no class root node, edges are not oriented) from a program binary. Next binary checks based on paths formed by the object calling relationships can be inserted. In future we want to implement several algorithms similar to as the ones described above in order to reconstruct a quasi class hierarchy with high accuracy and use it in order to compute possible calltarget sets for each previously detected callsite. Finally, this sets can be superimposed on the sets determined by our currently available forward edge policy with the goal to further shrink the legitimate calltarget set per callsite.

XI. Conclusion

In this paper, we presented TypeShield, a runtime fine-grained CFI-policy enforcing tool which operates on program binaries. Our tool precisely and efficiently filters legitimate from illegitimate forward and backward indirect control flow transfers by using a novel runtime type-checking technique based on function parameter type-checking and parameter-counting. We have implemented TypeShield and applied it to real software such as web servers, FTP servers and the SPEC CPU2006 benchmark. We demonstrated through extensive experiments that TypeShield has higher precision w.r.t. the calltarget set per callsite than existing state-of-the-art tools, while maintaining a comparable runtime performance overhead of 4%. To date, we were able to improve the ratio of calltargets per callsite by more than 87% w.r.t. total number of AT calltargets, by up to 35% for a single hardened application w.r.t. the parameter count policy and with an overall geomean reduction of more than 13% when comparing with TypeArmor. Next to a more precise analysis, the tangible outcome is a considerably reduced attack surface which can be further improved by tweaking our analysis. Finally, in the spirit of open research, we have made the source code of TypeShield, test scripts and the evaluation results publicly available, thus we support reproducibility in this fast-moving research field by providing comprehensive descriptions of our experiments. https://github.com/domain/typeshield.

References

[1] U. S. NVD, “In total 700 arbitrary code executions reported, jan.’08 to may’17,” 2017, https://nvd.nist.gov/vuln/search/results?adv_search=true&form_type=advanced&results_type=overview&query=arbitrary+code+execution&pub_date_start_month=0&pub_date_start_year=2008&pub_date_end_month=6&pub_date_end_year=2017.

[2] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[3] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, “It’s a trap: Table randomization and protection against function-reuse attacks,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[4] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz, “Subversive-c: Abusing and protecting dynamic message dispatch,” in *USENIX Annual Technical Conference (USENIX ATC)*, 2016.

[5] “Bluelotus team, bctf challenge: bypass vtable read-only checks,” 2015.

[6] B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng, “Loop-oriented programming: A new code reuse attack to bypass modern defenses,” in *IEEE Trustcom/BigDataSE/ISPA*, 2015.

[7] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control flow integrity,” in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[8] ——, “Control flow integrity principles, implementations, and applications,” in *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[9] G. Ramalingam, “The undecidability of aliasing,” in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1994.

[10] D. Jang, T. Tatlock, and S. Lerner, “Safedispatch: Securing c++ virtual calls from memory corruption attacks,” in *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[11] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, and H. Bos, “Shrinkwrap: Vtable protection without loose ends,” in *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[12] D. Bounov, R. Gökhan Kici, and S. Lerner, “Protecting c++ dynamic dispatch through vtable interleaving,” in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[13] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing forward-edge control-flow integrity in gcc and llvm,” in *Proceedings of the USENIX conference on Security (USENIX SEC)*, 2014.

[14] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical control flow integrity & randomization for binary executables,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[15] M. Zhang and R. Sekar, “Control flow integrity for cots binaries,” in *Proceedings of the USENIX conference on Security (USENIX SEC)*, 2013.

[16] A. Prakash, X. Hu, and H. Yin, “Strict protection for virtual function calls in cots c++ binaries,” in *Symposium on Network and Dist—ributed System Security (NDSS)*, 2015.

[17] C. Zhang, C. Song, K. C. Zhijie, Z. Chen, and D. Song, “vtint: Protecting virtual function tablesíntegrity,” in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[18] M. Elsabagh, D. Fleck, and A. Stavrou, “Strict virtual call integrity checking for c ++ binaries,” in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[19] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, “Marx : Uncovering class hierarchies in c++ programs,” in *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[20] V. v. d. Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, “A tough call: Mitigating advanced code-reuse attacks at the binary level,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P),*, 2016.

[21] “Itanium c++ abi,” https://mentorembedded.github.io/cxx-abi/abi.html.

[22] “C++ abi for the arm architecture,” 2015, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041e/IHI0041Ecppabi.pdf.

[23] “J. gray. c++: Under the hood,” 1994, http://www.openrce.org/articles/files/jangrayhood.pdf.

[24] “Bypassing clang’s safestack for fun and profit,” in *Blackhat*

Europe, 2016. [Online]. Available: https://www.blackhat.com/docs/eu-16/materials/eu-16-Goktas-Bypassing-Clangs-SafeStack.pdf

[25] W. Jansen, "Directions in security metrics research," in *NISTIR 7564*, 2009, http://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7564.pdf.

[26] I. JTC1/SC22WG21, "Iso/iec 14882:2013 programming language c++ (n3690)," 2013, https://isocpp.org/files/papers/N3690.pdf.

[27] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *Proceedings of the USENIX Conference on Security (USENIX SEC)*, 2015.

[28] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, "Vtrust: Regaining trust on virtual calls," in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[29] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: Theory and Practice*. CRC Press, 2009.

[30] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[31] "Dynamorio," http://dynamorio.org/home.html.

[32] R. Gawlik and T. Holz, "Towards automated integrity protection of c++ virtual function tables in binary programs," in *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[33] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2015.

[34] M. Conti, P. Larsen, S. Crane, L. Davi, M. Franz, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[35] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the USENIX conference on Security (USENIX SEC)*, 2015.

[36] D. Andriesse, X. Chen, V. v. d. Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proceedings of the USENIX Conference on Security (USENIX SEC)*, 2016.

[37] A. Mycroft, "Lecture notes," 2007, https://www.cl.cam.ac.uk/~am21/papers/sas07slides.pdf.

[38] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Symposium on Network and Distributed System Security (NDSS)*, 2010.

[39] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Proceedings of Computer AidedVTint: Protecting Virtual Function Tab Verification (CAV)*, 2011.

[40] A. Fokin, Y. Derevenets, A. Chernov, and K. Troshina, "Smartdec: Approaching c++ decompilation," in *Working Conference on Reverse Engineering (WCRE)*, 2011.

[41] G. Balakrishnan and T. Reps, "Divine: Discovering variables in executables," in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2007.

[42] J. Caballero and Z. Lin, "Type inference on executables," in *ACM Computing Surveys (CSUR)*, 2016.

[43] "Intel control-flow enforcement technology," 2016, https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[44] "Windows control flow guard," 2015. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

[45] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, "Vtpin: Practical vtable hijacking protection for binaries," in *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[46] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[47] V. v. d. Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensiticve cfi," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[48] C. t. F. i. M. W. X. S. P. . Microsoft, https://technet.microsoft.com/en-us/library/bb457151.aspx.

[49] P. T. A. S. L. Randomization, 2001, https://pax.grsecurity.net/docs/aslr.txt.

[50] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[51] T. Kornau, "Return-oriented programming for the arm architecture," 2009, http://www.zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf.

[52] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[53] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2011.

[54] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[55] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Return-oriented programming without returns on arm," in *Technical report, Technical Report HGI-TR-2010-002, Ruhr-University Bochum*, 2010.

[56] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *Proceedings of the USENIX conference on Security (USENIX SEC)*, 2014.

[57] M. Zhang and R. Sekar, "Control flow and code integrity for cots binaries: An effective defense against real-rop attcks," in *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[58] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Symposium on Network and Dist—ributed System Security (NDSS)*, 2015.

[59] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptograhically enforced control flow integrity," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[60] A. Prakash and H. Yin, "Defeating rop through denial of stack pivot," in *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[61] T. Petsios, V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Dynaguard: Armoring canary-based protections against brute-force attacks," in *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[62] P. Yuan, Q. Zeng, and X. Ding, "Hardware-assisted fine-grained code-reuse attack detection," in *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.

[63] B. Niu and G. Tan, "Modular control-flow vtint: Protecting virtual function tabintegrity," in *ACM Conferece on Programming Language Design and Implementation (PLDI)*, 2014.

[64] ——, "Rockjit: Securing just-in-time compilation using modular control-flow inegrity," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[65] ——, "Per-input control-flow integrity," in *Proceedings the ACM Conference on Computer and Communications Security (CCS)*, 2015.