TYPESHIELD: Practical Forward & Backward Edge Defense Against Code Reuse Attacks

***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 41% compared to state-of-the-art tools.

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

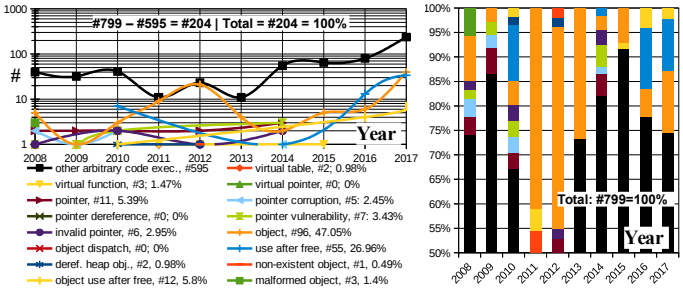

Fig. 1: Arbitrary code reuse attacks vs. object corruptions.

this class of attacks is the ability to corrupt a virtual object pointer in order to call gadgets by using a list of fake objects.

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 forward-edges and backward-edges. Unfortunately, COOP and its brethren bypass most deployed CFI-based enforcement policies, since these attacks do not exploit indirect backward-edges (*i.e.,* function returns), but rather exploit the forward indirect control flow transfers (*i.e.,* object dispatches) 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 virtual 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]. X axis is years (left & right) and Y axis is number of reports in logarithmic scale (left, 204) and distribution in % of the reports (right, 799). In black are the ACE unrelated reports. As of August'17, U.S. NVD reports in total 799 ACEs from which 204 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 §VI-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-edge and backward-edge filtering, that significantly reduces the number valid forward and backward targets 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 carefully analyzing function parameter counts and related register characteristics. 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 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 strict calltarget set per callsite compared to other state-of-the-art tools. TYPESHIELD uses automatically inferred parameter types which are used to build a more precise approximation of both the callee parameter types and callsite signatures. This is later used during the classification of matching callsites and calltargets, in order to distinguish between valid and invalid function calls, and results in a more precise callee target set for each caller than other state-of-the-art approaches like, for instance, TypeArmor.

Analysis Details. The TYPESHIELD analysis is based on a use-def callees analysis to derive 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 backward-edge policy is available in three modes: (1) super fast mode, a range is imposed for each AT function return formed by the minimum and maximum legitimate addresses, (2) fast mode, each AT function return contains a label value which is compared against several return targets labels, and (3) slow mode, each AT function return address is compared individually with each legitimate return address target where the calltarget (*i.e.,* function return) wants to return to. This backward-edge policy modes represent effective fine-grained SafeStack [24] (*i.e.,* compiler based protection recently bypassed [25].) alternatives.

Comparison. TYPESHIELD uses different basic block analysis strategies than TypeArmor, and no control flow graph as TypeArmor does. Further, TYPESHIELD disallows a forward indirect call transfer where the types of the arguments provided are not super types (*i.e.,* the float type is a super type for int) 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 return site, thus drastically reducing the attacker leeway.

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 [26] and the security and measurements section in [27]) by releasing TYPESHIELD open source and by providing for each conducted experiment a precise description.

II. Background

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 [28]), because it can be exploited to call: (x) illegitimate virtual table entries (not) contained in the class hierarchy by potentially varying the number of parameters and types, (y) legitimate virtual table entries (not) contained in the class hierarchy by potentially varying the number of parameters and types, and (z) fake virtual tables entries not contained in the class hierarchy by potentially varying 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, see (x) above). 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 and parameter number, we insert before each indirect callsite a check used for determining during runtime if the caller matches with the callee based on certain CFI policies.

B. Checking Indirect Calls in Practice

To the best of our knowledge, only the IFCC/VTV [13] compiler based tools (up to 8.7% performance overhead) are deployed in practice and can be used to check legitimate from illegitimate indirect forward-edge calls during runtime. Virtual pointers 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 (≈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 [29] (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] (around 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. Overview an Threat Model

In this section, we present in §III-A our type based policy and approach overview in Figure 2. In §III-B we highlight our backward edge protection policy, while in §III-C we introduce our threat model.

A. Type Based Policy

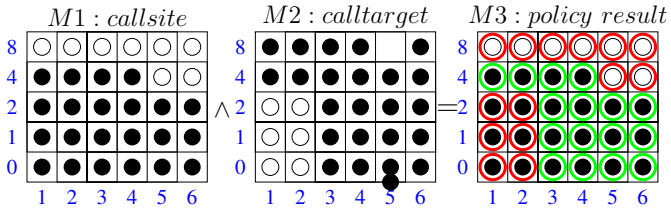

Fig. 3: Forward indirect edges parameter type & count policy.

Figure 3 depicts on the X axis (parameter count) and Y axis (register wideness) of matrices $M1$, $M2$ and $M3$ which 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. Further, Figure 3 highlights the behavior of our type based policy when the callsite provides

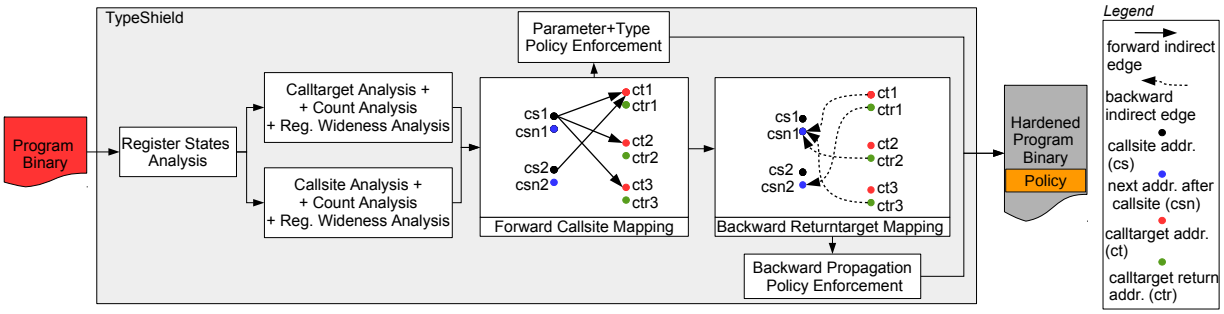

Fig. 2: Approach overview. The original program binary is analyzed (see left hand side above Figure) by TYPESHIELD and the calltargets and callsite analysis are performed for determining how many parameters are provided, how many are consumed and their register wideness. After this step labels are inserted at each previously identified callsite and at each calltarget. The enforced policy is schematically represented by the black highlighted dots (addresses) in the above Figure which are allowed to call only legitimate red highlighted dots (addresses). Next for each function return address the address set determined by each address located after each legitimate (is allowed to call the function) callsite is collected. This information is obtained by using the previously determined callsite forward-edge mapping to derive a function return backward map containing function returns as key and return targets as values. In this way TYPESHIELD has for each function return site a set of legitimate addresses where the function return site is allowed to transfer the program control flow. Finally, range or compare checks are inserted before each function return site. This checks are used to check during runtime if the address where the function return wants to jump to is contained in the legitimate set for each particular return site. This is represented above Figure by green highlighted dots (addresses) that are allowed to call only legitimate blue highlighted dots (addresses). Finally, the result is a hardened program binary (see right hand side above Figure).

6 parameters $\langle pcs1, ..., pcs6 \rangle$ having following bit wideness $pcs1$: 4-byte, $pcs2$: 4-byte, $pcs3$: 4-byte, $pcs4$: 8-byte, $pcs5$: 2-byte, $pcs6$: 2-byte, and the calltarget is expecting 6 parameters $pct1, ..., pct6$ having following bit wideness $pct1$: 4-byte, $pct2$: 4-byte, $pct3$: 0-byte, $pct4$: 0-byte, $pct5$: 0-byte, $pct6$: 0-byte of the expected parameters. TypeArmor's parameter count policy is the following.

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

The forward-edge policy of TYPESHIELD is as follows.

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

However, one can observe that the first policy (Definition 1) offers less precision w.r.t. forward edge mapping on the legitimate target set than the second policy (Definition 2). Note that the first policy performs only parameter count checks whereas the second policy checks for each parameter index in part separately w.r.t. count and register wideness.

B. Backward Edge Policy

TYPESHIELD uses a backward edge (*i.e.,* `retn`) fine-grained CFI protection policy which relies on enforcing the legitimate forward edge addresses after each callsite to each calltarget return address (*i.e.,* function return address). This corresponds to the caller-callee calling convention which basically enforces that each function should return to the next address after the callsite which was used to call that function before. TYPESHIELD provides three modes of operation for protecting the backward-edge policy. This modes of operation will be presented in section §IV.

C. 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. DESIGN

In this section, we present in §IV-A we present our function parameter count based policy, in §IV-B, we present the details of our type policy, and in §IV-C we introduce the definitions for our instructions analysis based on register states, in §IV-D we present the design of our calltarget analysis, while in §IV-E we depict the design of our callsite analysis[2]. Finally, in §IV-F we present our forward edge policy instrumentation strategy, and in §IV-G we highlight our function backward edge analysis and policy instrumentation strategy.

A. Parameter Count Based Policy

Figure 4 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

[2]Callsites detection in the binary is based on DynInst [32] capabilities.

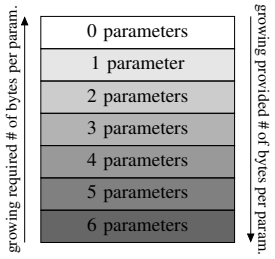

Fig. 4: Callsite & calltargets count policy classification.

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

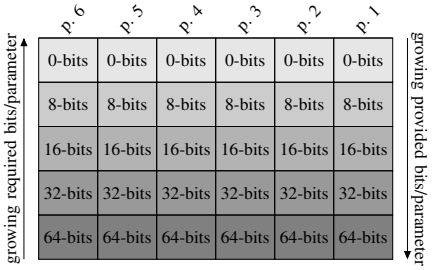

Fig. 5: Type policy schema for callsites and calltargets.

Figure 5 depicts the basic principle of our function parameter type policy. 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.

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 $\texttt{INSTR}$ 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 $i \in \texttt{INSTR}$ 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 $i \in \texttt{INSTR}$ via the function $decode : \texttt{INSTR} \mapsto \Delta^{16}$.

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 iterates over address-taken[3] functions with the goal of analyzing register state information in order to determine if these registers are used for arguments passing.

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.

[3] A program function is defined to have its address taken if there is at least one binary instruction which loads the function entry point into memory. Note that by definition, indirect calls can only target AT functions.

Algorithm 1: Basic block liveness analysis.

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

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

Algorithm 1 is based on the liveness analysis algorithm presented in [30], 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) Void/Non-Void Calltarget: In order to determine if a calltarget is a void or non-void return function TYPESHIELD traverses backwards the basic blocks from the return instruction of the function an looks for the `RAX` register. In case there is a write operation on the `RAX` register than TYPESHIELD infers that the function return is non-void and thus provides a pointer value back.

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.

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 is based on the reaching definition analysis presented in [30], 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.

Algorithm 2: Basic block reaching definition analysis.

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

```
1  Function analyze(block : INSTR*) : S^R is
2      state = ⊥    ▷ Initialize the state with first block
3      foreach inst ∈ reversed(block) do
4          state' = analyze_instr(inst)    ▷ Calculate changes
5          state = merge_v(state, state')    ▷ Merge changes
6      end
7      states = ∅    ▷ Set of predecessor states
8      blocks = pred(block)    ▷ Get predecessors blocks
9      foreach block' ∈ blocks do
10         state' = analyze(block')    ▷ Analyze pred. block
11         states = states ∪ {state'}    ▷ Add pred. states
12     end
13     state' = merge_h(states)    ▷ Merge predecessors states
14     return merge_v(state, state')    ▷ Merge to final state
15 end
```

• `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 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 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 [30] 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) Void/Non-Void Callsite: In order to determine if a callsite is a void or non-void return function TYPESHIELD looks at the callsite if there is an read before write on the `RAX` register. In case there is a read before write operation on the `RAX` register than TYPESHIELD infers that the callsite is non-void and thus expects a pointer to be provided when the called function returns.

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* $\in$ $fMap$ **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* $\in$ $calltargetSet$ **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
      $rMap$ = $rMap \cup rMap\ add\ (rAddress, rTarget)$
    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 §IV-D and §IV-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 for a function return is determined by getting the next address after each callsite address which is allowed to make the forward edge control flow transfer (*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 basic block 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.,* [31]) exist, and plan to experiment with them in 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 (calltarget) return present in the $rMap$. We propose three modes of operation based on three types of checks which can be inserted before each function return instruction. Depending on the specific needs one of the following modes of operation can be selected.

Super fast mode. Based on the $rMap$, for each AT 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 $rAddresse$s. In short, this means that as far as the min and max addresses are from each other the more leeway the attacker will have.

Fast mode. Based on the $rMap$, before each AT function return a randomly generated label (*i.e.,* the value 7232943 loaded trough one level of indirection) value will be inserted. The same label will be inserted before each legitimate (*i.e.,* based on the forward-edge policy) target address (next address after a legitimate callsite) of the function return. In this way a function return will be allowed to jump to only the instruction which follows next to the address of the callsites which are allowed to call the calltarget which contains this particular function return. For callsites which are allowed to call the the calltarget mentioned and another calltarget than in this cases TypeShield will perform a search in order to detect if the callsite has already a label attached to the next address after the callsite. In this case the label will be reused. In this situation two callsites share their labels. The solution to this is to use single labels for each function return address. In this case multiple labels have to be stored for each address following a legitimate callsite. Further, addresses located after a callsite that are not allowed to call a particular calltarget will get another randomly generated label. In this way calltarget return labels are grouped together based on the $rMap$. This mode allows at least (additionally the callsites which are allowed to call more than one calltarget are added) the same number of function return sites as the forward-edge policy enforces for

each callsite and it is runtime efficient since label checking is based on a single compare check.

Slow mode. Based on the $rMap$, before each AT 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.

V. Implementation

We implemented TypeShield using the DynInst [32] (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 [33] 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.

VI. Evaluation

We evaluated TypeShield by instrumenting various open source applications and conducting a thorough analysis. Our test sample includes the two FTP servers *Vsftpd* (v.1.1.0, C), Pure-ftpd (v.1.0.36, C) and *Proftpd* (v.1.3.3, C), web server Lighttpd (v.1.4.28, 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).

RQ1: How **precise** is TypeShield? (§VI-A)
RQ2: How **effective** is TypeShield? (§VI-B)
RQ3: What **overhead** imposes TypeShield? (§VI-C)
RQ4: What **binary blow-up** has TypeShield? (§VI-D)
RQ5: What **security level** offers TypeShield? (§VI-E)
RQ6: Which **upper bounds** has TypeShield? (§VI-F)
RQ7: Which **attacks** mitigates TypeShield? (§VI-G)
RQ8: Is TypeShield **effective** against COOP? (§VI-H)
RQ9: Are other tools **better** than TypeShield? (§VI-I)
RQ10: Is shadow-stack **better** than TypeShield? (§VI-J)

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 into 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 be able to better compare the obtained results with other existing 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 (RQ1)

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)
MySQL	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: 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. 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. 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.

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.

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.

2) Count Based Classification Precision:

−O2	Calltargets			Callsites		
Target	#cs gt	perfect args	perfect return	#ct gt	perfect args	perfect return
ProFTPd	1009	898 (88.99%)	845 (83.74%)	157	130 (82.8%)	113 (71.97%)
Pure-FTPD	128	107 (83.59%)	52 (40.62%)	8	4 (50%)	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%)
MySQL	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.

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. The #cs gt and #ct gt labels mean total number of callsites and calltarget based on the ground truth, respectively. 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 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:

-O2	Calltargets			Callsites		
Target	#cs gt	perfect args	perfect return	#ct gt	perfect args	perfect return
Proftpd	1009	835 (82.75%)	861 (85.33%)	157	125 (79.61%)	113 (71.97%)
Pure-Ftpd	128	101 (78.9%)	54 (42.18%)	8	4 (50%)	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 #cs gt and #ct gt labels mean total number of callsites and calltarget based on the ground truth, respectively. 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 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.

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 (RQ2)

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.

O2 Target	AT	*count** limit (mean ± σ)	*count** median	*count* limit (mean ± σ)	*count* median	*type** limit (mean ± σ)	*type** median	*type* limit (mean ± σ)	*type* median
ProFTPD	396	330.31 ± 48.07	343.0	334.5 ± 51.26	311.0	310.58 ± 60.33	323.0	337.41 ± 54.09	336.0
Pure-FTPD	13	5.5 ± 4.82	6.5	9.87 ± 4.32	13.0	4.37 ± 4.92	2.0	8.12 ± 4.11	7.0
Vsftpd	10	7.14 ± 1.81	6.0	7.85 ± 1.39	7.0	5.42 ± 0.95	6.0	6.42 ± 0.96	7.0
Lighttpd	63	27.75 ± 10.73	24.0	41.19 ± 13.22	41.0	25.1 ± 8.98	24.0	41.42 ± 14.29	38.0
MySQL	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.

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 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 (RQ3)

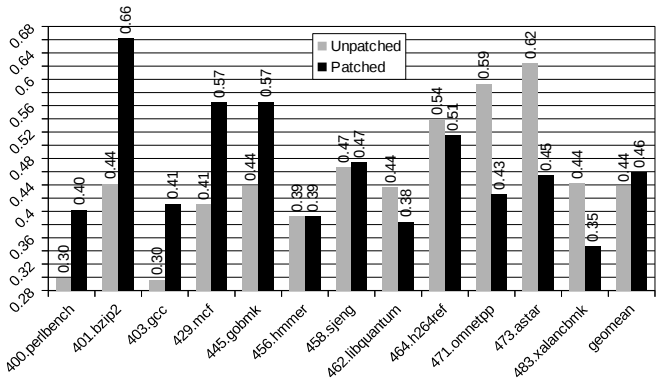

Fig. 6: SPEC CPU2006 Benchmark Results.

Figure 6 depicts the runtime overhead obtained by applying TYPESHIELD (forward-edge policy (parameter count and type mode) and backward-edge policy (fast mode)) to several SPEC CPU2006 benchmarks. Unpatched means the original vanilla programs while patched means the programs with the CFI checks inserted. The obtained runtime overhead is around 4% (geomean) when instrumenting the binary with 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 [34] (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 (RQ4)

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 (RQ5)

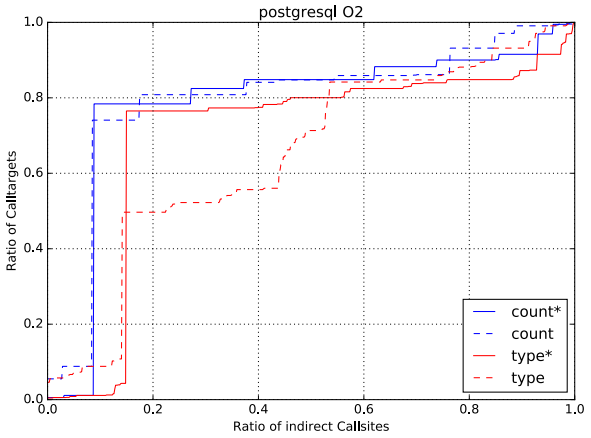

Fig. 7: CDF for PostgreSQL compiled with Clang `-O2`.

Figure 7 depicts the cumulative distribution function (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 (*i.e.,* this is the number of equivalence classes) 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 7 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 (RQ6)

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 theoretical 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. Mitigation of Advanced Code-Reuse Attacks (RQ7)

In this section we briefly list all code reuse attacks types that can be stopped by TYPESHIELD. We want to remark that basically all CFI violating attacks can be mitigated by TYPESHIELD. Non-control flow violating attacks (*i.e.,* data-only attacks) are out of scope.

Exploit	Stopped	Remark
COOP ML-G [2]		
IE 32 bit	×	Out of scope
IE 1 64-bit	✓(FP)	Argcount mismatch
IE 2 64-bit	✓(FP)	Argcount mismatch
Firefox	✓(FP)	Argcount mismatch
COOP ML-REC [3]		
Chrome	✓(FP)	Void target where non-void was expected
Control Jujutsu [35]		
Apache	✓(FP)	Target function not AT
Nginx	✓(FP)	Void target where non-void was expected
All Backward edge violating attacks	✓(BP)	(1)[a] or (2)[b] or (3)[c]

[a] Jump to an address that is $\notin$ in the $max - min$ address range.
[b] Jump to an address which is $\neq$ to one of the legitimate addresses.
[c] Jump to an address label that is $\neq$ with the calltarget return label.

TABLE V: Stopped attacks.

Table V depicts several attacks that can be successfully stopped by TYPESHIELD by deploying only the forward-edge or the backward-edge policy. In Table V the FP label means forward-edge policy while BP means backward-edge policy. In summary, all forward-edge based attacks and backward edge attacks can be successfully mitigated by TYPESHIELD as long these attacks are not aware of the policy in place and thus can not selectively use gadgets which have their start address in the allowed set for the legitimate forward-edge and backward-edge transfers, respectively.

H. Effectiveness Against COOP (RQ8)

We investigated the effectiveness of TYPESHIELD against the COOP attack by looking at the number of register arguments which can be used to enable data-flow between gadgets. In order to determine how many arguments remain unprotected after we apply the forward edge policy of TYPESHIELD we compared the number of parameter overestimation and compare it with the ground truth obtained with the help of an LLVM compiler pass. Next we used some heuristics to determine how many ML-G and REC-G callsites exist for each of the C++ only server applications. Finally, we compared these results with the one obtained by TypeArmor.

		Overestimation					
Program	#cs	0	+1	+2	+3	+4	+5
MySQL (ML-G)	192	184	3	1	0	1	3
Node.js (ML-G)	134	131	1	0	1	0	1
geomean	160	155	1	1	1	1	1
MySql (REC-G)	289	273	10	2	3	0	1
Node.js (REC-G)	72	69	2	0	0	0	1
geomean	144	137	4	1	1	1	1

TABLE VI: Parameter overestimation for ML-G and REC-G.

Table VI depicts the results obtained after counting the number of perfectly and overestimation of protected ML-G and REC-G gadgets. As it can be observed we obtained a 96% (184 vs. 192) accuracy (geomean) of perfectly protected ML-G callsites for MySQL while TypeArmor obtaines for the same program an 94% accuracy (geomean). Further, TYPESHIELD obtained a 97% (131 vs. 134) accuracy (geomean) for Node.js while TypeArmor obtained 95% accuracy on the same program. Further, for the REC-G case TYPESHIELD obtained an 94% (273 vs.289) exact argument accuracy for MySQL while TypeArmor had 86%. For Node.js TYPESHIELD obtained an exact parameter matching of 95% (69 vs. 72) while TypeArmor obtained an 96% perfect matching.

Overall TYPESHIELDs forward edge policy obtained an perfect accuracy of 95% while TypeArmor obtained 92%. While this is not a big difference we point out that the remaining overestimated parameters represent 5% and this do not leave much room for the attacker to perform her attack.

I. Forward-Edge Comparison with Other Tools (RQ9)

Target	IFCC	TypeArmor (CFI+CFC)	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 VII: Calltargets per callsite reduction statistics.

Table VII 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. 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. TYPESHIELD reduces the number of calltargets by up to 35% (geomean) when compared to the AT functions, by up to 41% (12 vs. 7) for a single test program and by 13% (geomean) when comparing with TypeArmor, respectively. Finally, TYPESHIELD represents a strong improvement w.r.t. calltarget per callsite reduction in binary programs.

J. Backward-edge Comparison with Shadow-Stack (RQ10)

There are several tools which provide shadow-stack protection as for example the original shadow-stack implementation from Abadi [7](binary-based), SafeStack [24] (compiler based and bypassed [25]), MoCFI [36] (compiler based), HAFIX [37] (compiler based and bypassed [38]), and also PathArmor [39] (similarly by-passable as kBouncer, ROPecker, and ROP-Guard [40] as it relies on a capacity restricted LBR register.) which emulates a shadow stack trough validation of the last-branch register (LBR). Further, there are many compiler based approaches which can be used to protect backward edges.

The only binary based shadow-stack approach is the one from Abadi et al. [7]. This solution has: (1) a high rumtime overhead ($\geq$ 21%), (2) is not open source, (3) uses a proprietary binary analysis framework (*i.e.,* Vulcan), and (4) uses a restricted number of labels, *i.e.,* each function called from

inside a function will get the same label. This label will be stored in all function shadow stacks, see Figure 1 in [7].

For this reason we propose an alternative backward-edge protection solution which is more runtime efficient. In order to show the precision of TYPESHIELD backward edge protection we will give the average number of legitimate return addresses for each calltarget return address and relate it to the total number of available addresses without any protection.

Program	Total #RA	Total #RATs	Total #RATs/RA	%RATs/RA w.r.t. prog. binary
MySQL	5896	3792	0.64	0.014%
Node.js	7230	3864	0.53	0.011%
geomean	6529	3827	0.58	0.012%

TABLE VIII: Backward-edge policy statistics.

Table VIII depicts the statistics w.r.t. the backward edge policy legitimate return targets. In Table VIII we use the following abbreviations: total number of return addresses (Total #RA), total (median) number of return address targets (Total #RATs), total (median) number of return addresses targets per return addresses (Total. # RATs/RA), percent of legitimate return address targets per return addresses w.r.t. the total number of addresses in the program binary (% RATs/RA w.r.t. program binary). By applying TYPESHIELD backward-edge policy we obtain a reduction of 0.43 $(1-0.58)$ ratio (geomean) of total number of return addresses targets per return addresses over total number of return addresses which means that only 43% of the total number of return addresses are actual targets for the function returns. The results indicate a percentage of 0.012% (geomean) of the total addresses in the program binaries are legitimate targets for the function returns. This means that our policy can eliminate 99.98% (100% - 0.012%) of the addresses which an attacker can use for his attack inside the program binary.

VII. RELATED WORK

A. Mitigation of Forward-Edge based Attacks.

1) Binary Based Techniques: TypeArmor [20] is a binary instrumentation tool that can protect against COOP. 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.

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 protect against backward edge violations and arguably has in common with VCI several limitations.

B. Mitigation of Backward-Edge based Attacks.

Backward-Edge based code reuse attacks exploit the indirection provided by the return instructions of a function. Usually each modern compiler builds caller/callee pairs by adhering to the so called caller-callee calling convention. This calling convention basically specifies that for each indirect call the return address of the function which returns after it was called lies at the next address of the call instruction. This calling convention is violated by all ROP attacks and also by more recent advanced code reuse attacks. Intel CET [41] is a promising technology from Intel in which the X86 instruction set is updated with new instructions (*i.e.,* `END_BRANCH`) instruction which should facilitate an efficient implementation of shadow stack implementations. Currently, this technology is not available and it is not foreseeable when this features will be available in mass production. For brevity reasons we do not look herein into compiler and purely runtime techniques and advise the reader to look for more details the following survey [42].

1) Binary Based Techniques: The CFI based implementation of Abadi et al. [8] is historically the first compiler based implementation of a shadow stack. At first quite promising this implementation suffers from high performance overhead which is around 21% due to the fact that the inserted checks before each function return instruction are not runtime efficient.

VIII. 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, databases, 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 (forward-edge) by more than 87% w.r.t. total number of AT calltargets, by up to 41% for a single hardened application and with an overall geomean reduction of more than 13% when comparing with TypeArmor. Further,

by using our backward-edge policy, 99.98% (geomean) of the addresses from a binary are eliminated as potential function return targets. 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 will make 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.

REFERENCES

[1] U. S. NVD-Reports, "Over 800 arbitrary code executions reported from jan.'08 to may'17," 2017, https://goo.gl/5uv47n.

[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] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "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 Distributed 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] 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.

[25] "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

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

[27] C. Herley and P. C. van Oorschot, "Science, security, and the elusive goal of security as a scientific pursuit," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

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

[29] 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.

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

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

[32] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, (PASTE)*, 2011.

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

[34] 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.

[35] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskosr, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[36] B. Niu and G. Tan, "Modular control-flow integrity," in *ACM Conferece on Programming Language Design and Implementation (PLDI)*, 2014.

[37] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: hardware-assisted flow integrity extension," in *Design Automation Conference (DAC)*, 2015.

[38] M. Theodorides and D. Wagner, "Breaking active-set backward-edge cfi," in *International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017.

[39] 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.

[40] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Evaluating the effectiveness of current anti-rop defenses," in *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2014.

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

[42] B. Nathan, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," in *ACM Computing Surveys (CSUR)*, 2018.