

TYPESHIELD: Practical Defense Against Code Reuse Attacks using Binary Type Information

tba.

Abstract—We propose TYPESHIELD, a binary runtime forward-edge and backward-edge protection tool used for instrumenting program executables during load time. TYPESHIELD enforces a novel runtime control-flow integrity (CFI) policy based on function parameter type and count, in order to overcome the limitations of available approaches and to efficiently verify dynamic object dispatches and function returns during runtime. To enhance practical applicability, TYPESHIELD can be automatically and easily used in conjunction with legacy applications or where source code is missing to harden binaries. We evaluated TYPESHIELD on highly relevant open source programs and the SPEC CPU2006 benchmark and were able to efficiently and with low performance overhead protect these applications from forward-edge and backward-edge corruptions. Finally, in a direct comparison with state-of-the-art tools, TYPESHIELD achieves higher caller-callee matching precision, while maintaining a low runtime overhead.

I. INTRODUCTION

The C++ programming language offers object-oriented programming (OOP) concepts that are highly relevant during development of large, complex and efficient software systems, in particular, when runtime performance and reliability are primary objectives. A key OOP concept is polymorphism, which is based on C++ virtual functions. Virtual functions 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 virtual table meta-data structures for all virtual functions and provide to each instance (object) of such a class a (virtual) pointer (its value is computed during runtime) to the aforementioned table. While this approach represents a main source for exploitable program indirection (*i.e.*, forward-edges) along function returns (*i.e.*, backward-edges) the C/C++ language provides no intrinsic security guarantees (*i.e.*, we consider Clang-CFI [7] and Clang’s SafeStack [2] optional).

Our work is primarily motivated by the absence of source code and by the presence of at least one exploitable memory corruption (*e.g.*, buffer overflow, etc.), which can be used to enable the execution of sophisticated Code-Reuse Attacks (CRAs) that can violate control flow graph (CFG) forward-edges of the program’s CFG such as the advanced COOP attack [36] and its extensions [15], [28], [5], [26] and/or the backward-edge such as Control Jujutsu [18]. A potential ingredient for violating forward-edge control flow transfers is based on corrupting a virtual object pointer while backward-edges can be corrupted by loading fake return addresses on the program stack in order to consecutively call gadgets. To address such object dispatch corruptions and in general any type

of indirect program control flow transfer violation, Control-Flow Integrity (CFI) [8], [9] was originally developed to secure indirect control flow transfers by adding runtime checks before forward-edges and backward-edges. While CFI-based techniques that rely on the construction of a precise CFG are effective [13] and, in general, if CFGs are carefully constructed and sound [37] these techniques cannot effectively be used if a CRA does not violate the previously constructed CFG-based policy. For example, the COOP family of CRAs bypass most deployed CFI-based enforcement policies, since these attacks do not exploit indirect backward-edges (*i.e.*, function returns), but rather CFG forward-edge imprecision (*i.e.*, object dispatches, indirect control flow transfers), which in general cannot be statically (before runtime) precisely determined as alias analysis in program binaries is undecidable [35] and certain program CFG edges are input dependent.

Source code based tools which can protect against forward-edge violations such as: SafeDispatch [23], ShrinkWrap [22], VTI [12], and IFCC/VTV [38] rely on source code availability which limits their applicability (*i.e.*, proprietary libraries cannot be recompiled). In contrast, binary-based forward-edge protection tools, binCFI [42], vfGuard [34], vTint [41], VCI [17], Marx [32] and TypeArmor [40], typically protect only the forward-edges through a CFI-based policy and most of the tools assume that a shadow stack [25] technique for protecting backward-edges is in place.

Unfortunately, the currently most precise binary based forward-edge protection tools w.r.t. calltarget reduction, VCI and Marx, suffer from forward edge imprecision since both are based on an approximated program class hierarchy (*i.e.*, no root class determined, and the edges between the classes are not oriented) derived through the usage of up to six heuristics and several simplification assumptions, while TypeArmor enforces a forward-edge policy which takes into account only the number of parameter provided and consumed by caller-callee pairs without imposing any constraint on their types. Thus, these forward-edge protection tools are in general too permissive. As there is clear evidence that CFI-based forward-edge protection techniques without a backward-edge protection are broken [14], these tools further assume that a shadow stack protection policy is in place. Unfortunately, recently shadow stack based techniques (backward-edge protection, on average up to 10% runtime overhead [16]) were bypassed [20], [1]. Further, this bypass demonstrated that at least 4 independently usable attack vectors (namely: guessing oracles, neglected pointers, thread spraying, and allocation oracles) exist for thwarting shadow stack techniques (binary and source

code based) targeting their entropy based hiding principle and making their usage questionable.

In this paper, we present TYPESHIELD, a fine-grained CFI-complete (*i.e.*, forward-edge and backward-edge protection) runtime binary-level protection tool which does not rely on shadow stack based techniques to protect backward-edges. We implemented TYPESHIELD on top of DynInst [11], which is a binary rewriting framework that allows program binary instrumentation during loading or runtime. TYPESHIELD does not rely on runtime type information (RTTI) (*i.e.*, metadata emitted by the compiler, most of the time stripped in production binaries) or particular compiler flags, and is applicable to legacy programs. TYPESHIELD takes a not obfuscated program binary as input and it automatically instruments it in order to detect illegitimate indirect control flow transfers during runtime. In order to achieve this, TYPESHIELD analyzes program binaries (works on x86 and x86-64) by carefully analyzing function parameter register wideness (used to derive parameter type information) and the provided and consumed number of function parameters. Based on the used ABI, TYPESHIELD is consequently able to track up to 6 function arguments for the Itanium C++ ABI [4] (used by the major Unix OSs, *i.e.*, Linux, BSD, and MacOSX) x86-64 calling convention. The Itanium ABI caller-callee calling convention essentially means that every called function will return at the next address located after the callsite which was used in first place to call this function. This means that there is a one-to-one mapping (we exclude tail calls) between each caller and callee contained in the program. However, we stress that the presented technique is applicable with the ARM ABI [6] and Microsoft’s C++ ABI [21] as well.

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 for all calltargets, (3) compares individually parameters of callsites and calltargets in order to check if an indirect call transfer is legitimate or not, and (4) based on the provided forward-edge caller-callee mapping it builds a mapping back from each callee to the legitimate addresses located next to each caller, thereby providing a more strict callsite per calltarget compared to other state-of-the-art tools and a fine-grained shadow stack alternative for backward edges. TYPESHIELD’s backward-edge policy is based on the observation that backward-edges of a program can be efficiently protected if there is a precise forward-edge mapping available between callers and callees. TYPESHIELD uses automatically inferred parameter types, which are used to build a more precise approximation of both the callee parameter types and callsite signatures.

We evaluated TYPESHIELD with several highly real-world open source programs (*i.e.*, NodeJS, Lighttpd, MySQL, etc.) and the SPEC CPU2006 benchmark and show that our forward-edge policy is more precise than state-of-the-art and our backward-edge policy is a precise alternative to shadow stacks. TYPESHIELD is applicable to program binaries for which we assume source code is not available. TYPESHIELD

significantly reduces the number of valid forward-edges compared to previous work [40] and thus we are able to build a precise backward-edge policy which represents an efficient alternative to shadow stack based techniques. Thus, there is no need to assume, as other forward edge protection techniques, that another backward-edge protection mechanism (*i.e.*, shadow stack) is in place. In this way, the attack vectors of shadow stack are avoided.

In summary, we make the following main contributions:

PM: precisely state what are the novel contributions.

- We designed a novel fine-grained CFI technique for protecting forward-edges and backward-edges without making any assumptions on the presence of a shadow stack based technique.
- We implemented, TYPESHIELD, a binary instrumentation prototype which enforces a fine-grained forward-edge and backward-edge protecting technique for stripped program binaries. TYPESHIELD can serve as platform for developing other binary based protection mechanisms.
- We conducted a thorough set of evaluative experiments in which, we show that TYPESHIELD is more precise and effective than other state-of-the-art tools. Further, we show that our tool has a higher calltarget set per callsite reduction, thus further reducing the attack surface compared to state-of-the-art tools.

II. BACKGROUND AND RELATED WORK

In the following, we provide a brief overview of the technical concepts we use in the rest of this paper to detect and constrain forward and backward edges in program binaries as well as related work and our threat model.

A. Security Implications of Indirect Transfers

Indirect Forward-Edge Transfers. Forbidden forward-edge indirect calls are the result of a virtual pointer (vPointer) corruption. A vPointer corruption is not a vulnerability, but rather a capability which can be the result of a spatial or temporal memory corruption triggered by: (1) bad-casting [27] of C++ objects, (2) buffer overflow in a buffer adjacent to a C++ object or (3) a use-after-free condition [36]. A vPointer corruption can be exploited in several ways. A manipulated vPointer can be exploited by pointing it in any existing or added program virtual table entry or into a fake virtual table added by an attacker. For example, in case a vPointer was corrupted, an attacker could hijack the control flow of the program and start a COOP attack [36].

vPointer corruptions are a real security threat that can be exploited if there is a memory corruption (*e.g.*, buffer overflow), which is adjacent to the C++ object or a use-after-free condition. As a consequence, each corruption, which can reach an object (*e.g.*, bad object casts), is a potential exploit vector for a vPointer corruption.

Indirect Backward-Edge Transfers. Program backward edges (*i.e.*, jump, ret, etc.) can be corrupted due to their provided indirection to assemble gadget chains in the following scenarios. (1) No CFI protection technique was applied: In this

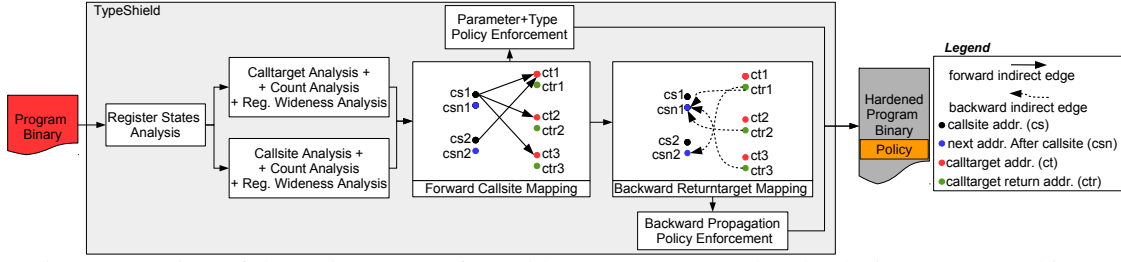


Fig. 1: Overview of the main steps performed by TYPESHIELD when hardening a program binary.

case, the binary is not protected by any CFI policy. Obviously, the attacker can then hijack backward edges to *jump* virtually anywhere in the binary in order to chain gadgets together. (2) Coarse-grained CFI protected scenarios: In this scenario, if the attacker is aware of what addresses are protected and which are not through control flow bending, the attacker may deviate the application flow to legitimate locations in order to link gadgets together. (3) Fine-grained CFI protection scenarios: In this case, the legitimate target set is more strict than in (2) but still in this case, under the assumption that the attacker knows which addresses are protected and which are not, she may be able through control flow bending to call legitimate targets, which are useful for assembling a gadget chain. (4) Fully precise CFI protected scenarios (*i.e.*, SafeStack [25] based): In this scenario, the legitimate target set is more strict than in (3). Even though, we have a one-to-one mapping between calltargets and legitimate return sites, the attacker could theoretically use this one-to-one mapping to assemble gadget chains if at the legitimate calltarget return site there is a useful gadget. See [14] for more details about how control flow can be bent to legitimate addresses and how this is still dangerous in case that at the bent address location exists a useful gadget even in fully CFI-complete scenarios.

B. Mitigation of Forward-Edge Based Attacks

Amongst **binary based tools**, TypeArmor [40] enforces a CFI-policy based on runtime checking of caller-callee pairs which relies on function parameter count matching. Compared to TYPESHIELD, this tool does not use function parameter types and assumes that a backward-edge protection is in place. VCI [17] and Marx [32] are both based on approximated program class hierarchies: (1) do not recover the root class of the hierarchy, and (2) the edges between the classes are not oriented. Thus, both tools enforce for each callsite the same virtual table entry (*i.e.*, index based) contained in one recovered class hierarchy denoted by father-child relationships between the recovered vttables. Finally, both tools use up to six heuristics and simplifying assumptions in order to make the problem of program class hierarchy reconstruction tractable.

C. Mitigation of Backward-Edge Based Attacks

According to one of the currently most comprehensive surveys by Burow *et al.* [13] assessing backward edge protection techniques and runtime overhead comparisons, tools can be distinguished into providing low, medium, and high levels of protection w.r.t. backward-edges.

Specifically for **binary based tools**, the survey authors provide the following insights. The original CFI implementation from Abadi *et al.* [8] as well as MoCFI [29], kBouncer [31], CCCFIR [42], bin-CFI [43], O-CFI [30], PathArmor [39], LockDown [33] mostly suffer from imprecision (high number of reused labels); have low runtime efficiency; and some of the tools do not support shared libraries.

D. Threat Model

We align our threat model with the same basic assumptions as described in [40] w.r.t. the forward-edge. More precisely, we assume a resourceful attacker that has read and write access to the data sections of the attacked program binary. We 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 exploit an existing memory corruption in order to hijack the program control flow. As such, we consider a powerful, yet realistic adversary model that is consistent with previous work on code-reuse attacks and mitigations [25]. The adversary is aware of the applied defenses and has access to the source and non-randomized binary of the target application. He can exploit (bend) any backward-edge based indirect program transfer and has the capability to make arbitrary memory writes. We assume that other forward-edge and backward-edge protection mechanisms can be used in parallel with our techniques. These defense mechanisms are orthogonal to our protection policies. Our approach does not rely on information hiding from the attacker and as such we can tolerate arbitrary reads. Finally, the analyzed program binary is not hand-crafted and the compiler which was used to generate the program binary adheres to one of the following most used caller-callee function calling conventions [6], [21], [4].

III. SYSTEM OVERVIEW

In §III-A, we present the main steps performed by TYPE-SHIELD in order to harden a program binary, and in §III-B, we introduce the invariants for calltargets and callsites, while, in §III-C, we describe our backward edge protection policy.

A. Approach Overview

Figure 1 depicts the overview of our approach. From right to left, the program binary is analyzed (see left hand side in Figure 1) 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, *e.g.*, *cs1*) in Figure 1 which are allowed to call only legitimate red highlighted dots (addresses *e.g.*, *ct1*). 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 computed. 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. These 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 in Figure 1 by green highlighted dots (addresses *e.g.*, *ctr2*) that are allowed to call only legitimate blue highlighted dots (addresses *e.g.*, *csn1*). Finally, the result is a hardened program binary (see right hand side in Figure 1).

B. Invariants

1) *Calltargets and Callsites*: We propose the following invariants for the function calltargets and callsites. (1) indirect callsites provide a number of parameters (*i.e.*, possibly overestimated compared to program source code), (2) calltargets require a minimum number of parameters (*i.e.*, possibly underestimated compared to program source code), and (3) the wideness of the callsite parameters has to be larger or equal to the wideness of the parameters registers expected at the calltarget. In a nutshell, the idea is that a callsite might only call functions that do not require more parameters than provided by the callsite and where the parameter register wideness of each parameter of the callsite is higher or equal to that parameter register used at the calltarget. Figure 1 depicts this requirements by the forward indirect edges pointing from the black dots to the legitimate red dots *e.g.*, *ct2*.

2) *Calltarget Returns*: We propose the following invariant for the calltargets returns: we enforce the caller-callee convention between the calltarget return instruction and the address next to the callsite, which was used in the first place to call that calltarget. Figure 1 depicts this with the backward indirect edges pointing from the green shaded dots to the legitimate blue shaded dots *e.g.*, *csn2*.

C. 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 enforces that each function is allowed to return to the next

address after the callsite which was used to call that function first.

IV. DESIGN AND IMPLEMENTATION

In §IV-A, the details of our register type analysis is presented, and in §IV-B, we introduce the register state analysis, while in §IV-C, we present the design of our calltarget analysis. In §IV-D, we depict the design of our callsite analysis¹, and in §IV-E, we present our backward edge analysis, while in §IV-F, we present our forward-edge and backward-edge policy instrumentation strategies. Finally, in §IV-G, we present implementation details.

A. Parameter Register Wideness Based Policy

We use the register width of the function parameters in order to infer the type information. There are 4 types of reading and writing accesses. Therefore, our set of possible types for parameters is $\text{TYPE} = \{64, 32, 16, 8, 0\}$; where 0 models the absence of a parameter. Since the 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 TYPE^6 . We allow overestimations of callsites and underestimations of calltargets on the parameter types. Therefore, for a callsite *cs* it is possible to call a calltarget *ct*, only if for each parameter of *ct* the corresponding parameter of *cs* is not smaller w.r.t. the register width. This results in a fine-grained policy, which is further restricting the possible set of calltargets for each callsite.

Based on this information, we can assign a register one of 5 possible types $\mathcal{T} = \{64, 32, 16, 8, 0\}$. We also included the type 0 to model the absence of data within a register. However, the matching idea is different, since the type of a calltarget and a callsite no longer depend solely on its parameter count, each callsite and calltarget has its type from the set of \mathcal{T}^6 , with the following comparison operator: $u \leq_{\text{type}} v \iff \forall_{i=0}^5 u_i \leq v_i$, with $u, v \in \mathcal{T}^6$.

We allow any callsite *cs* to call any calltarget *ct*, when it fulfills the requirement $ct \leq cs$. We represent this by letting the type for a calltarget parameter progress from 64-bit to 0-bit—if a calltarget requires a 32-bit value in its 1st parameter, it also should accept a 64-bit value from its callsite—and similarly, we let the type for a callsite progress from 0-bit to 64-bit. If a callsite provides a 32-bit value in its 1st parameter it also provides a 16-bit, 8-bit and 0-bit to a calltarget. The advantage of the *type* policy in comparison to a parameter *count* only policy is that while our type comparison implies the count comparison, the other direction does not hold. Meaning, just having an equal or lesser number of parameters than a callsite, does no longer allow a calltarget being called there, thus restricting the number of calltargets per callsite even further. For example, a function which requires 64-bit in its first parameter, and 0-bit in all other parameters, would have been callable by a callsite providing 8-bit in its first and second parameter when using a parameter *count* policy, however in the

¹Callsites detection in the binary is based on the capabilities of DynInst.

type policy this is no longer possible. Thus, this will decrease the number of targets per bucket.

Along our *type* analysis, we built a function parameter count-based policy similar to [40]. Calltargets are classified based on the number of parameters that these provide and callsites are classified by the number of parameters that these require. Further, we consider the generation of high precision measurements for such classification with binaries as the only source of information rather difficult. Therefore, over-estimations of parameter count for callsites and under-estimations 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 [4]—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 $\forall cs_1, cs_2; c_{cs_1} \leq c_{cs_2} \rightarrow \|\{ct \in \mathcal{F} \mid c_{ct} \leq c_{cs_1}\}\| \leq \|\{ct \in \mathcal{F} \mid c_{ct} \leq c_{cs_2}\}\|$.

Thus, within this paper the overall reduction of callsites, which can access the same set of calltargets, will be explored. Yet another research path is to introduce an upper bound for the classification deviation of parameter counts, however, as of now, this will be left for future work. Overall the count policy complements the type policy with the goal to further reduce the legitimate calltarget set per callee.

B. Analysis of Register States

Our register state analysis is (as the name implies) register state based. Another alternative would be to do symbol-based data-flow analysis, which we will leave as future work. In order for the reader to understand our analysis we will first give some definitions. The set INSTR describes all possible instructions that can occur within the executable section of a program binary. In our case, this is based on the x86-64 instruction set. An instruction $i \in \text{INSTR}$ 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. The Itanium 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 $i \in \text{INSTR}$ via the function $\text{decode} : \text{INSTR} \mapsto \Delta^{16}$.

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

The set \mathcal{I} describes all possible instructions that can occur within the executable section of a binary. In our case, this is based on the instruction set for x86-64 processors. Thus, an instruction $i \in \mathcal{I}$ can non-exclusively perform two kinds of operations on any number of existing registers: (1) Read n -bit from the register with $n \in \{64, 32, 16, 8\}$, and (2) Write n -bit to the register with $n \in \{64, 32, 16, 8\}$.

The Itanium C++ ABI specifies 16 general purpose integer registers, thus for our purpose we represent the change occurring at the processor level as $\mathcal{S} = \mathcal{S}^{16}$. At last, we declare a function, which calculates the change occurring in the processor state, when executing an instruction from \mathcal{I} : $\text{decode} : \mathcal{I} \mapsto \mathcal{S}$.

C. 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 (AT) functions with the goal of analyzing register state information in order to determine if these registers are used for arguments passing. Note 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.

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

Algorithm 1 is based on the liveness analysis algorithm presented by Khedker *et al.* [24]. This algorithm consists of 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.

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

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

Algorithm 1: Basic block liveness analysis.

Input : The basic block to be analyzed - block : INSTR*
Output: The liveness state - S^L

```

1 Function analyze (block : INSTR*) :  $S^L$  is
2   state = BI           ▷ 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_v(states) ▷ Merge succ. states
14  return merge_v(state, state') ▷ Merge to final state
15 end

```

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

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

In our specific case, the function `analyze_instr` needs to also handle non-jump and non-fall-through successors, as these are not handled by `DynInst`. Essentially, there are three relevant cases. First, if the current instruction is an indirect call or a direct call and the analysis algorithm is set not to 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 analysis algorithm is set not to follow calls, then we start an analysis of the target function and 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 S^L .

2) *Required Parameter Wideness*: For our type policy, we use a fine 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 an access. This gives us the following $S^L = \{C, r8, r16, r32, r64, W\}$ register state which translates to the register super state $S^L = (S^L)^{16}$. As there could be more than one read of a register before it is written, we are 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 we 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 vertical merge merge_v 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.

3) *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 an access. Further, this gives us the $S^L = \{C, R, W\}$ as register state, which translates to the register super state $S^L = (S^L)^{16}$. 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 $\text{merge}(\text{merge_h})$ 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.

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 and looks for the `RAX` register. In case there is a write operation on the `RAX` register, then `TYPESHIELD` infers that the function return is non-void and thus provides a pointer value back.

D. Callsite Analysis

Our callsite analysis classifies callsites according to the parameters they provide. Similar to the calltarget set analysis, overestimations are allowed, while, underestimations are not permitted. For this purpose, we employ a customizable modified reaching definition algorithm, which we present next.

1) *Reaching Definitions*: An assignment to a variable is a reaching definition after the execution of a set of instruction if this 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 by Khedker *et al.* [24]. This analysis can be regarded as a reverse depth-first traversal of program basic blocks. For customization, we rely on the implementation of several functions. S^R is the set of possible register states which depends on the specific reaching definition implementation of the following operations.

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

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

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

Algorithm 2: Basic block reaching definition analysis.

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

```

1 Function analyze( $block : INSTR^*$ ) :  $S^R$  is
2    $state = BI$            ▷ Initialize the state with first block
3   foreach  $inst \in reversed(block)$  do
4      $state' = analyze\_instr(inst)$    ▷ Calculate changes
5      $state = merge\_v(state, state')$  ▷ Merge changes
6   end
7    $states = \emptyset$            ▷ Set of predecessor states
8    $blocks = pred(block)$        ▷ Get predecessors blocks
9   foreach  $block' \in blocks$  do
10     $state' = analyze(block')$      ▷ Analyze pred. block
11     $states = states \cup \{state'\}$  ▷ 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

```

• $pred : INSTR^* \mapsto \mathcal{P}(INSTR^*)$, (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 analysis algorithm is set not to follow calls, then it returns a state where all registers are considered trashed, (2) if the instruction is a direct call and the analysis algorithm is set to follow calls, then we start an analysis of the target function, and (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 S^R .

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

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 an union like operator with T preceding S , and S preceding U . Note, that when both states are set, we pick the higher one.

3) *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 $S^L = \{T, S, U\}$ register state, which translates to the register super state $S^R = (S^R)^{16}$.

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 , then 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.

4) *Void/Non-Void Callsite*: As in the case of the calltarget analysis, in order to determine if a callsite is a void or non-void return function, TYPESHIELD analyzes the callsite in order to determine if there is a read before write on the RAX register. In case there such an operation on the RAX register then, TYPESHIELD infers that the callsite is non-void and thus expects a pointer to be provided when the called function returns.

E. 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 for each callee.

Algorithm 3: Calltarget return set analysis.

Input : Forward edge callsite to calltargets map - $fMap$
Output: Backward edge to return addresses map - $rMap$

```

1 Function backwardAddressMapping( $fMap$ ) :  $rMap$  is
2   ▷ visit all detected callsites in the binary
3   foreach  $callsite \in fMap$  do
4     ▷ get calltargets for callsite address key
5      $calltargetSet = getCalltargetSet(callsite, fMap)$ 
6     ▷ calltarget is the function start address
7     ▷ visit all calltargets of a callsite
8     foreach  $calltarget \in calltargetSet$  do
9       ▷ get the next address after the callsite
10       $rTarget = getNextAddress(callsiteKey)$ 
11      ▷ find the address of function return
12       $rAddress = getReturnOfCalltarget(calltarget)$ 
13      ▷ rAddress is map key; rTarget is value
14       $rMap = rMap \cup rMap.add(rAddress, rTarget)$ 
15    end
16  end
17  ▷ return the backward-edgeaddresses mappings
18  return  $rMap$ 
19 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-C and §IV-D for more details). This mapping contains for each callsite the legal calltargets where the forward-edge indirect control flow transfer is allowed to jump to. This mapping is mapped back by building a second mapping between the return address of each function

(callee) 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 the 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.*, [10]) exist, and plan to experiment with them in future work.

F. Binary Instrumentation

1) *Forward-Edge Policy Enforcement*: 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 these 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, then the transfer is allowed, else the program execution is terminated.

2) *Backward-Edge Policy Enforcement*: The previously determined *rMap* in Algorithm 3 will be used to insert a check before each function (calltarget) return present in the *rMap*. For enforcing the this policy, we propose a runtime mode of operation based on a single CFI check which will be inserted before each function return instruction.

Based on the previously determined *rMap*, before each AT function return a randomly generated label (*i.e.*, the value 7232943 will be loaded through 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 containing

this particular function return. For callsites which are allowed to call the calltarget mentioned and another calltarget, 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. Note that 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 of operation 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 efficient compare check. Finally, in case the compare returns true, then the execution continues, else it is terminated.

G. Implementation

We implemented TYPESHIELD using the DynInst [11] (v.9.2.0) instrumentation framework. In total, we implemented TYPESHIELD in 5501 lines of code (LOC) of C++ code. We currently restricted our analysis and instrumentation to x86-64 bit executable and linking format (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 [3] 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 an Clang/LLVM (v.4.0.0, trunk 283889) back-end (machine instruction) pass (416 LOC) used for collecting ground truth data in order to better assess the effectiveness 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 contained in 3239 LOC of Python code.

V. EVALUATION

We evaluated TYPESHIELD by instrumenting various open source applications and conducting a thorough analysis in order to show its effectiveness and usefulness. Our test sample includes the two FTP servers *Vsftpd* (v.1.1.0, C code), *Pure-ftpd* (v.1.0.36, C code) and *Proftpd* (v.1.3.3, C code), web server *Lighttpd* (v.1.4.28, C code); the two database server applications *Postgresql* (v.9.0.10, C code) and *Mysql* (v.5.1.65, C++ code), the memory cache application *Memcached* (v.1.4.20, C code), and the *Node.js* application server (v.0.12.5, C++ code). We selected these applications in order

to allow for a fair comparison with [40]. In our evaluation, we addressed the following research questions (RQs).

RQ1: How **precise** is TYPESHIELD? (§V-A) **RQ2:** How **effective** is TYPESHIELD? (§V-B) **RQ3:** Are other forward-edge tools **better** than TYPESHIELD? (§V-C) **RQ4:** Is TYPESHIELD **better** than ShadowStack? (§V-D) **RQ5:** What **overhead** imposes TYPESHIELD? (§V-E)

Experimental Setup. Our setup used 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 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) machine instruction pass inside the x86-64 bit code generation implementation of LLVM. During compilation, we essentially collected 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 Target	Calltargets			Callsites		
	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)
Postgresql	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)
<i>geomean</i>	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 calltargets 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 TYPESHIELD’s callsites is comparable.

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 showing 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 TYPESHIELD’s calltargets is very good.

Callsites. While 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) Type Based Classification Precision:

O2 Target	#cs gt	Calltargets		#ct gt	Callsites	
		perfect args	perfect return		perfect args	perfect return
Proftpd	1009	835 (82.75%)	861 (85.33%)	157	125 (79.61%)	113 (71.97%)
Pure-Ftpd	128	101 (78.9%)	54 (42.18%)	8	4 (50%)	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%)
MySql	9728	6141 (63.12%)	7684 (78.98%)	8002	4477 (55.94%)	6449 (80.59%)
Postgresql	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%)
<i>geomean</i>	1097.06	867.43 (79.06%)	723.70 (65.96%)	203.77	139.08 (68.25%)	180.59 (88.62%)

TABLE II: *type* based policy classification of callsites.

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. We used 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.

Calltargets. For the first experiment, we used the union combination operator with an *analyze* function that follows 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)

Table III depicts the average number of calltargets per callsite, the standard deviation σ and the median. We evaluated the effectiveness of TYPESHIELD by leveraging the results of several experimental runs. First, we established a baseline using the data collected from our Clang/LLVM pass. These are the theoretical limits of our implementation, which can be reached for both the count and the type schema. Second, we evaluated the effectiveness of our count policy. Third, we evaluated the effectiveness of our type policy.

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 experimental 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 II), 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 II). 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 experimental 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 II). 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 II). 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 the total number of AT calltargets available after our *count* and *type* policies were applied.

C. Forward-Edge Policy vs. Other Tools (RQ3)

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
<i>geomean</i>	7.6	162.1	216.6	172.0	141.8

TABLE IV: Calltargets per callsite reduction statistics.

Table IV 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. As such, TYPESHIELD represents a strong improvement w.r.t. calltarget per callsite reduction in binary programs.

D. Comparison with Shadow-Stack (RQ4)

The safe stack implementation of Abadi *et al.* [8] has the highest security level [13] w.r.t. backward-edge protection. This solution has: (1) a high runtime overhead ($\geq 21\%$), (2) is not open source, (3) uses a proprietary binary analysis framework (*i.e.*, Vulcan), (4) reuses a restricted number of labels, *i.e.*, each function called from inside a function will get the same label, and (5) the shadow stacks can be disclosed by a

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

TABLE III: Results for allowed callsites per calltarget for several programs 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.

determined attacker. These labels will be stored in all function shadow stacks, see Figure 1 in [8].

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 V: Backward-edge policy statistics.

Table V depicts the statistics w.r.t. the backward-edge policy legitimate return targets. In Table V, 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’s 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. This 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. Thus, only 0.012% of the addresses inside the binary can be used as return addresses by the attacker.

Further, we assume that knowing exactly which addresses are still available is hard to determine by the attacker for any given program binary which is stripped from debug information. Note, that each function return (callee) is allowed to return to around 150 legitimate addresses for the analyzed program. Finally, we assume that it is hard for the attacker to find out exactly these per return site legitimate addresses after the policy was applied.

E. Runtime Overhead (RQ5)

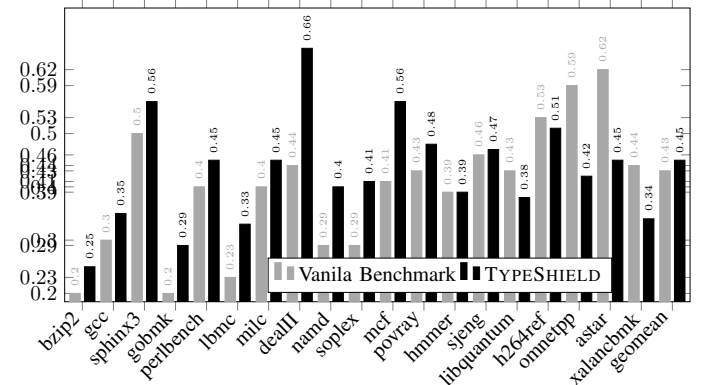


Fig. 2: Benchmark run time normalized against the baseline for the SPEC CPU2006 benchmarks.

PM: add rest of the spec benchmarks. Figure 2 depicts the runtime overhead obtained by applying TYPESHIELD (forward-edge policy (parameter count and register type) and backward-edge policy) to several programs contained in the SPEC CPU2006 benchmark. Out of the evaluated programs xalanbmk, astar, and omnetpp, dealII, namd, soplex, and povray are C++ programs, while the rest are pure C programs. Unpatched means the original vanilla programs, while patched means the programs with the forward-edge and backward-edge CFI checks inserted. After the programs were instrumented,

we measured the runtime overhead. The obtained runtime overhead is around 3.29% (geomean, 0.43 vs. 0.45) when instrumenting the binary with DynInst and 2.46% (0.44 vs. 0.45) on average. 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, jumps into the old executable section and from there back to the new executable section occur. 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 (yet acceptable) performance overhead is due to our runtime policy which is more complex than that of other state-of-the-art tools.

Further, the runtime overhead of TYPESHIELD (3%) is comparable with other forward-edge protection tools such as: TypeArmor (3%) (binary), VCI [17] (7.79% overall and 10.49% on only the SPEC CPU2006 programs) (binary), vfGuard [34] (binary) (10% - 18.7%), T-VIP [19] (0.6% - 103%) (binary), SafeDispatch [23](source code) (2% - 30%), and VTV/IFCC [38] (8% - 19.2%) (source code). Thus, these results qualify TYPESHIELD as a highly practical tool.

VI. CONCLUSION

We presented TYPESHIELD, a tool which can protect forward-edges and backward-edges of stripped (*i.e.*, no RTTI information) program binaries without the need to make any assumptions on the presence of an auxiliary technique for protecting backward edges (*i.e.*, shadow stacks) as most of the CFI policy enforcing tools do. We evaluated TYPESHIELD with real open source programs and have shown that TYPESHIELD is practical and effective when protecting program binaries. Further, our evaluation reveals that TYPESHIELD can considerably reduce the forward-edge legal call target set, provide high backward-edge precision, while maintaining low runtime overhead.

REFERENCES

- [1] Bypassing clang's safestack for fun and profit. In *Blackhat Europe'16*. <https://www.blackhat.com/docs/eu-16/materials/eu-16-Goktas-Bypassing-Clangs-SafeStack.pdf>.
- [2] Clang's safestack. <https://clang.llvm.org/docs/SafeStack.html>.
- [3] Dynamorio. <http://dynamorio.org/home.html>.
- [4] Itanium c++ abi. <https://mentoreembedded.github.io/cxx-abi/abi.html>.
- [5] Bluelotus team, btcf challenge: bypass vtable read-only checks. 2015.
- [6] C++ abi for the arm architecture. 2015. <http://infocenter.arm.com/help/topic/com.arm.doc.ih0041e/IHI0041Ecppabi.pdf>.
- [7] Clang cfi. 2017. <https://clang.llvm.org/docs/ControlFlowIntegrity.html#cfi-strictness>.
- [8] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control flow integrity. In *CCS'05*.
- [9] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control flow integrity principles, implementations, and applications. In *TISSEC'09*.
- [10] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *EuroS&P'17*.
- [11] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *PASTE'11*.
- [12] D. Bounov, Rami Gökhan K., and S. Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *NDSS'16*.

- [13] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. In *CSUR'17*.
- [14] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX SEC'15*.
- [15] 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 *CCS'15*.
- [16] T. H. Y. Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *ASIACCS'15*.
- [17] M. Elsabagh, D. Fleck, and A. Stavrou. Strict virtual call integrity checking for c++ binaries. In *ASIACCS'17*.
- [18] 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 *CCS'15*.
- [19] R. Gawlik and T. Holz. Towards automated integrity protection of c++ virtual function tables in binary programs. In *ACSAC'14*.
- [20] E. K. Goktas, A. Oikonomopoulos, Robert Gawlik, Benjamin Kollenda, I. Athanasopoulos, C. Giuffrida, G. Portokalidis, and H.J. Bos. Bypassing clang's safestack for fun and profit.
- [21] J. Gray. C++: Under the hood. 1994. <http://www.openrce.org/articles/files/jangrgrayhood.pdf>.
- [22] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, and H. Bos. Shrinkwrap: Vtable protection without loose ends. In *ACSAC'15*.
- [23] D. Jang, T. Tatlock, and S. Lerner. Safedispach: Securing c++ virtual calls from memory corruption attacks. In *NDSS'14*.
- [24] U. Khedker, A. Sanyal, and B. Sathe. *Data flow analysis: Theory and Practice*. CRC Press'09.
- [25] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI'14*.
- [26] 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'15*.
- [27] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX SEC'15*.
- [28] J. Lettner, B. Kollenda, A. Homescu, Per Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz. Subversive-c: Abusing and protecting dynamic message dispatch. In *USENIX ATC'16*.
- [29] D. Lucas, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *NDSS'12*.
- [30] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS'15*.
- [31] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX SEC'13*.
- [32] 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 *NDSS'17*.
- [33] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *DIMVA'15*.
- [34] A. Prakash, X. Hu, and H. Yin. Strict protection for virtual function calls in cots c++ binaries. In *NDSS'15*.
- [35] G. Ramalingam. The undecidability of aliasing. In *TOPLAS'94*.
- [36] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming. In *S&P'15*.
- [37] G. Tan and T. Jaeger. Cfg construction soundness in control-flow integrity. In *PLAS'17*.
- [38] 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 *USENIX SEC'14*.
- [39] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *CCS'15*.
- [40] V. van der 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 *S&P'16*.
- [41] C. Zhang, C. Song, K. Chen Zhijie, Z. Chen, and D. Song. vntint: Protecting virtual function tables integrity. In *NDSS'15*.
- [42] 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 *S&P'13*.

- [43] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *USENIX SEC'13*.