

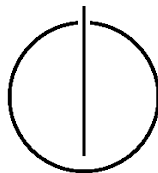
FAKULTÄT FÜR INFORMATIK

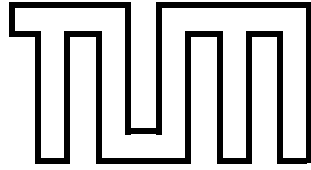
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

**Runtime protection against advanced
code reuse attacks by static
instrumentation of binaries**

Matthias Konstantin Fischer





FAKULTÄT FÜR INFORMATIK

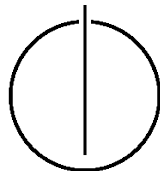
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Runtime protection against advanced code reuse
attacks by static instrumentation of binaries

Laufzeitschutz gegen fortgeschrittene Code
Wiederverwendungs-Angriffe durch statische
Instrumentierung von Binärdateien

Author: Matthias Konstantin Fischer
Supervisor: Prof. Dr. Claudia Eckert
Advisor: M.Sc. Paul Muntean
Date: November 7th, 2016



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, November 7th

Matthias Konstantin Fischer

Acknowledgments

If someone contributed to the thesis... might be good to thank them here.

Abstract

High security, high performance and high availability applications such as the Firefox and Chrome web browsers are implemented in C++ for modularity, performance and compatibility to name just few reasons. Virtual functions, which facilitate late binding, are a key ingredient in facilitating run-time polymorphism in C++ because it allows an object to use general (its own) or specific functions (inherited) contained in the class hierarchy. However, because of the specific implementation of late binding, which performs no verification in order to check where an indirect call site (virtual object dispatch through virtual pointers (vptrs)) is allowed to call inside the class hierarchy, this opens a large attack surface which was successfully exploited by the COOP attack. Since manipulation (changing or inserting new vptrs) violates the programmer initial pointer semantics and allows an attacker to redirect the control flow of the program as he desires, vptr corruption has serious security consequences similar to those of other data-only corruption vulnerabilities. Despite the alarmingly high number of vptr corruption vulnerabilities, the vptr corruption problem has not been sufficiently addressed by the researchers.

In this paper, we present *TypeShield*, a run-time vptr corruption detection tool. It is based on executable instrumentation at load time and uses a novel run-time type and function parameter counter technique in order to overcome the limitations of current approaches and efficiently verify dynamic dispatching during run-time. In particular, *TypeShield* can be automatically and easily used in conjunction with legacy applications or where source code is missing. It achieves higher caller/callee matching (precision) and with reasonable run-time overhead. We have applied *TypeShield* to real life software such as web servers, JavaScript engines, FTP servers and large-scale software including Chrome and Firefox browsers, and were able to efficiently and with low performance overhead to protect these applications from vptr corruptions vulnerabilities. Our evaluation revealed that *TypeShield* imposes up to X% and X% overhead for performance-intensive benchmarks on the Chrome and Firefox browsers, respectively.

@Matthias: add final % values at the end, when the evaluation is done.

Contents

Abstract	v
1 Introduction	1
1.1 Motivation	2
1.2 Contribution	3
1.3 Outline	3
2 Background: Forward C++ Forbidden Calls Exposed	5
2.1 Polymorphism in C++	5
2.2 Checking Indirect Forward-Edge Calls in Practice	7
2.3 Security Implications of Forbidden Indirect Calls	8
2.4 Real COOP Attack Example	9
3 TypeShield Overview	11
3.1 1. Select Important Point from Design Chapter	11
3.2 2. Select Important Point from Design Chapter	11
3.3 3. Select Important Point from Design Chapter	11
3.4 Adversary Model	11
3.5 TypeShield: Invariants for Targets and Callsites	12
3.6 TypeShield Impact on COOP	13
4 Design	15
4.1 <i>count</i> Policy	15
4.2 <i>type</i> Policy	16
4.3 Instruction Analysis	18
4.4 Calltarget Analysis	18
4.4.1 Variable Liveness Analysis Theory	19
4.4.2 Required Parameter Count	21
4.4.3 Required Parameter Wideness	22
4.4.4 Variadic Functions	23
4.5 Callsite Analysis	25
4.5.1 Reaching Definitions Theory	25
4.5.2 Provided Parameter Count	27
4.5.3 Provided Parameter Wideness	29

4.6	Address Taken Analysis	31
4.6.1	Address Taken Targets	31
4.6.2	Binary Analysis	32
5	Implementation	33
6	Evaluation	35
6.1	Precision of TypeShield	35
6.1.1	Quality and Applicability of Ground Truth	36
6.1.2	Precision Calltarget Classification (COUNT)	39
6.1.3	Precision Callsite Classification (COUNT)	42
6.1.4	Precision Calltarget Classification (TYPE)	50
6.1.5	Precision Enhanced Callsite Classification (TYPE)	53
6.2	Effectiveness of TypeShield	56
6.2.1	Theoretical Limit for the Reduction of Calltargets per Call- site	56
6.2.2	TypeShield vs Theoretical Limit	56
6.2.3	Effect of our AddressTaken Analysis	59
6.3	Security Analysis of TypeShield	59
6.4	Mitigation of Advanced Code-Reuse Attacks	59
6.4.1	COOP	59
6.4.2	COOP Extensions	59
6.4.3	Pure Data-only Attacks	59
7	Discussion	63
7.1	Comparison with TypeArmor	63
7.1.1	Precision of Classification	63
7.1.2	Reduction of Available Calltargets	64
7.2	Discrepancies or Problems	64
7.3	Limitations of TypeShield	65
7.4	Possible Venues of Improvement	65
8	Related Work	67
8.1	Type-Inference on Executables	67
8.2	Mitigation of Code-Reuse Attacks	67
8.3	Mitigation of Advanced Code-Reuse Attacks	68
9	Future Work	71
10	Conclusion	73
	Acronyms	75

List of Figures	77
List of Tables	79
Bibliography	83

1 Introduction

Control-Flow Integrity (CFI) [26, 27] is one of the most used techniques to secure program execution flows against advanced Code-Reuse Attacks (CRAs).

Advanced CRAs such as COOP [3]

Present the virtual function concept in C++, What does it is good for and what security implications does it have?

Talk about the security implications of vptr corruptions and give some CVEs numbers here.

Briefly talk about source code based solutions which protect against vptr corruptions and n the end against COOP. Talk about SafeDispatch, ShrinkWrap, Bounov et al, IFCC/VTV etc.

Give a presentation of TypeArmor

TypeArmor [8] is a run-time based tool which enforces a fine-grained forward edge policy in executables based on caller/callee matching based on parameter count.

The introduction should answer this questions:

1.What is the problem? There are no mechanisms in C++ implemented which check during late binding (realized through virtual call sites, indirect call sites in binaries) that the target of an indirect call site is legitimate or illegitimate.

Specify The Problem statement. In this thesis we want to develop a tool which can mitigate one of the most dangerous attack which exploits the missing security checks from above.

2.What are the current solutions? There are three lines of defence against COOP attacks, source code based, binary based and runtime based (there is no real application which can really defend against).

TypeArmor [8] is the most similar tool to *TypeShield* and it used function parameter counting by enforcing a fine-grained policy on valid indirect caller/callee pairs inside a binary. The policy is checked during run-time by checking that the number of parameters which an indirect call site provides matches with the number of parameter the callee expects. This invariant helps to defend against COOP and Control Flow Jujutsu.

@Matthias: add some limitations of type Armor.

3.Where the solutions lack? TypeArmor lacks in precision w.r.t. caller/callee matching since it relies only on parameter counting.

4. What is your idea? Our insight is to enforce a fine-grained CFI policy by combining function parameter count, type matching. This offers higher precision than TypeArmot and probably higher performance than TypeArmor has since we add less checks in the binary. Thus checking fewer checks in the binary results in a better performance overhead than comparable solutions.

5. Contributions. In summary, we make the following contributions: See contribution section, down.

1.1 Motivation

this should be up to a din A4 page, example (remove afterwards):

@Matthias: remove and add your motivation

The development of various defense mechanisms, such as the introduction of data execution prevention (DEP) [18], stack smashing protection (SSP) [13], and address space layout randomization (ASLR) [42], have raised the bar for reliable exploit development significantly. Nevertheless, a dedicated attacker is still able to achieve code execution [33, 25]. Information leaks are utilized to counter ASLR and reveal the layout of the address space or other valuable information [40, 33]. To circumvent DEP the attackers have added code-reuse attacks, such as return-oriented programming (ROP) [39, 9, 26], jump-oriented programming (JOP) [16, 11, 5], and call-oriented programming (COP) [10], to their repertoire. Code-reuse attacks do not inject new code but chain together small chunks of existing code, called gadgets, to achieve arbitrary code execution. In response to this success, the defensive research was driven to find protection methods against code-reuse attacks. Some results of this research are kBouncer [32], ROPecker [12], EMET [19]/ROPGuard [21], BinCFI [53], and CCFIR [52]. These defenses incorporate two main ideas. The first is to enforce control-flow integrity (CFI) [1, 2]. With perfect CFI the control-flow can neither be hijacked by code injection nor by code-reuse [22]. However, the overhead of perfect CFI is too high to be practical. Therefore the proposed defense methods try to find the sweet spot between security and tolerable overhead. The second idea is to detect code-reuse attacks by known characteristics of an attack like a certain amount of gadgets chained together. All of those schemes defend attacks on the x86/x86-64 architecture. For other architectures the research is lacking behind [15]. Several generic attack vectors have been published by the offensive side to highlight the limitations of the proposed defense methods. Although single implementations can be disabled with common code-reuse attacks by exploiting a vulnerability in the implementation [14, 7], the generic ones rely on longer and more complex gadgets [22, 17, 37, 10, 23]. Since the

gadgets lose their simplicity by becoming longer it also becomes harder to find specific gadgets and chain them together. To our knowledge there is no gadget finder available to search CFI resistant gadgets.

1.2 Contribution

example remove afterwards:

The objective of this thesis is to develop a tool that assists the exploit developer at locating gadgets which circumvent CFI and heuristic checks. All changes to registers and memory space must be immediately evident to the exploit developer. For this reason, we analyze the gadgets and categorize every register and every memory write by a set of semantic definitions. We use an IL for the analysis to support different architectures without the effort of adjusting the algorithms to new architectures. Because of the high architecture coverage, VEX [44] is our choice for the IL. VEX is part of Valgrind, an instrumentation framework intended for dynamic use [43]. To use VEX statically pyvex [51] is utilized. We instrument the SMT solver Z3 [48] for the analysis itself. Unfortunately, there is no interface from VEX or pyvex to Z3 available. Hence, we develop a translation from VEX to Z3. A wrapper adds the architecture layouts to Z3, thus making Z3 suitable for symbolic execution and path constraint analysis of the gadgets. To sum up, our thesis makes the following contributions:

1. We develop a tool to discover CFI and heuristic check resistant gadgets in an architecture independent offline search.

2. We provide semantic definitions for a convenient search of the gadgets.

3. We provide a modular translation unit from VEX/pyvex to Z3, including a symbolic execution engine implemented for x86, x86-64, and ARM.

1.3 Outline

The remainder of this thesis is organized as follows. We start by giving an overview of how *TypeShield* is designed to mitigate COOP attacks. Chapter 2 gives an overview of bad C++ forward edge calls and its security implications. Chapter 3 contains a high level overview of *TypeShield*. Chapter 4 gives an overview of the techniques used in *TypeShield*. Chapter 5 presents briefly the implementation details of our tool. The *TypeShield* implementation is evaluated and discussed in Chapter 6 and Chapter 7, respectively. Chapter 8 surveys related work. Finally, Chapter 9 highlights several future steps and Chapter 10 concludes this thesis.

2 Background: Forward C++ Forbidden Calls Exposed

This Chapter presents in Section 2.1 a brief overview about polymorphism in C++. Section 2.2 gives an overview over usage of forward-edge indirect call sites in practice. Section 2.3 depicts security implications of forbidden forward-edge indirect call sites. Finally, Section 2.4 illustrates an indirect forward-edge call site corruption which has been successfully exploited through a COOP attack on the Firefox web browser.

2.1 Polymorphism in C++

Polymorphism along inheritance and encapsulation are the most used modern object-oriented concepts in C++. Polymorphism in C++ allows to access 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)* run-time (dynamic, is implemented with inheritance and virtual functions), *c)* ad-hoc (e.g., if the range of actual types that can be used is finite and the combinations must be individually specified prior to use), and *d)* parametric (e.g., if code is written without mention of any specific type and thus can be used transparently with any number of new types it is called parametric polymorphism). The first two are implemented through early and late binding, respectively. In C++, overloading concepts fall under the category of *c)* and Virtual functions; templates or parametric classes fall under the category of pure polymorphism. C++ provides polymorphism through: *i)* virtual functions, *ii)* function name overloading, and *iii)* operator overloading. In this paper, we will be concerned with dynamic polymorphism—based on virtual functions (10.3 and 11.5 in ISO/IEC N3690 [24])—because these can be exploited to call: *x)* illegitimate vTable entries not/contained in the class hierarchy by varying or not the number of parameters and types, *y)* legitimate vTable entries not/contained in the class hierarchy by varying or not the number of parameters and types, *z)* fake vTables entries not contained in the class hierarchy by varying or not the number of parameters and types. By legitimate and illegitimate vTable entries we mean

```
[ framesep=2mm,      linenos,
frame=none, firstnumber=1, frame-
sep = 1.0cm, fontsize=, ]C++ class
nsMultiplexInputStream final :pub-
lic nsIMultiplexInputStream ,public
nsISeekableStream //A1 ,public nsIIPC-
SerializableInputStream //A2 ,public
nsICloneableInputStream //A3 nsTAr-
ray;nsCOMPtr;nsIInputStream; mStreams;
NS_METHODIMPnsMultiplexInputStream ::
Close()MutexAutoLocklock(mLock);mStatus = NS_BASE_S;
```

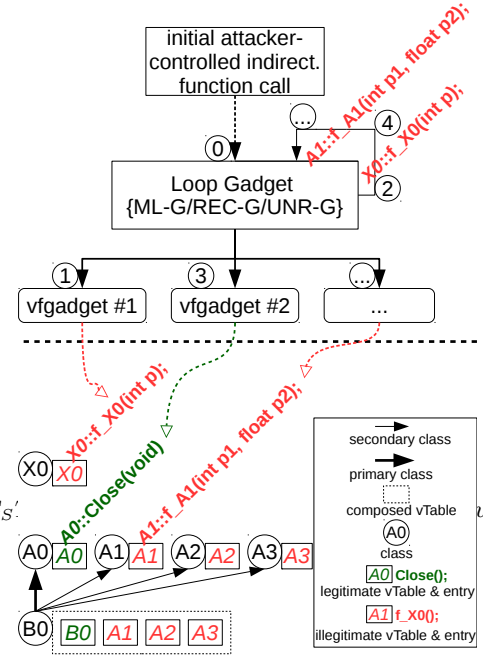


Figure 2.1: Code example used to illustrate how a COOP loop gadget works

those vTable entries which for a single indirect call site lie in the vTable hierarchy. More precisely, a vTable entry is legitimate for a call site if from the call site to the vTable containing the entry there is an inheritance path (see [9]). Virtual functions have several uses and issues associated, but for the scope of this paper we will look at the indirect call sites which are exploited by calling illegitimate vTable entries (functions) with varying number and type of parameters, x). More precisely, 1) load-time enforcement: as calling each indirect call site (callee) requires a fix number of parameters which are passed each time the caller is calling, we enforce a fine-grained CFI policy by statically determining the number and types of all function parameter that belong to an indirect call site. 2) run-time verification: as checking during run-time legitimate from illegitimate indirect caller/callee pairs requires parameter type (along parameter number), we check during run-time before each indirect call-site if the caller matches to the callee based on the previously added checks.

Figure 2.1 depicts a C++ code example where it is illustrated how a COOP loop gadget (ML-G, REC-G, UNR-G, see [13]) works. (1) can be exploited in several ways, see x), y) and z). The indirect call site (line 17) can be exploited to call by passing a varying number of parameters and types on each object contained in the array a different vTable entry contained in the: 1) class hierarchy

(overall, whole program), 2) class hierarchy (partial, only legitimate for this call site), 3) vTable hierarchy (overall, whole program), 4) vTable hierarchy (partial, only legitimate for this call site), 5) vTable hierarchy and/or class hierarchy (partial, only legitimate for this call site), and 6) vTable hierarchy and/or class hierarchy (overall, whole program). There is no language semantics—such as cast checks—in C++ for vCall sites dispatch checking and as consequence the loop gadget indicated in Figure 2.1 can basically call all around in the class and vTable hierarchy by not being constrained by any build in check during run-time. The attacker corrupts an indirect function call, ①, next she invokes gadgets, ①, ③, through the calls, ②, ④, contained in the loop. As it can be observed in in Figure 2.1 she can invoke from the same call site legitimate functions residing in the vTable inheritance path (this type of information is usually very hard to recuperate from executables) for this particular call site, indicated with green color vTable entries. However, a real COOP attack invokes illegitimate vTable entries residing in the whole initial program hierarchy (or the extended one) with less or no relationship to the initial call site, indicated with red color vTable entries.

2.2 Checking Indirect Forward-Edge Calls in Practice

As far as we know, there is only the IFCC/VTV [4] tools (up to 8.7% performance overhead) deployed in practice which can be used to check legitimate from illegitimate indirect forward-edge calls during compile time. vPointers are checked based on the class hierarchy. ShrinkWrap [9] (as far as we know not deployed in practice) is a tool which further reduces the legitimate vTbles ranges for a given indirect call site through precise analysis of the program class hierarchy and vTable hierarchy. Evaluation results show similar performance overhead but more precision w.r.t. to legitimate vTables entries per call site. 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 call site many range checks have to be performed during run-time. Therefore, despite its security benefit these types of checks can not be applied in our opinion to high performance applications.

As alternative, there are other highly promising tools (not deployed in practice) that can be used to mitigate some of the drawbacks of the previous tools. Bounov et al [11] presented a tool ($\approx 1\%$ runtime overhead) for indirect forward-edge call site checking based on vTable layout interleaving. The tool has better performance than VTV and better precision w.r.t. allowed vTables per indirect call site. Its precision (selecting legitimate vTables for each call site) compared to ShrinkWrap is lower since it does not consider vTable inheri-

tance paths. vTrust [12] (average runtime overhead 2.2%) enforces two layers of defense (virtual function type enforcement and vTable pointer sanitization) against vTable corruption, injection and reuse. TypeArmor [8] (\leq than 3 % runtime overhead) enforces an CFI policy based on runtime checking of caller/callee pairs based on function parameter count matching (coarse grained, parameter types and more than six parameters can be used as well). Important to notice is that there are no C++ language semantics which can be used to enforce type and parameter count matching for indirect call/callee pairs, this could be addressed with specifically intended language constructs in future.

2.3 Security Implications of Forbidden Indirect Calls

The C++ language standard (12.7 [24]) does not specify what happens when calling different vTable entries from an indirect call site. The standard says that we have a virtual function related undefined behavior when: “a virtual function call uses an explicit class member access and the object expression refers to the complete object of x or one of that object’s base class subobjects but not x or one of its base class subobjects”. As undefined behavior is not a clearly defined concept we argue that in order to be able to deal with undefined behavior or unspecified behavior related to virtual function calls one needs to know how these language dependent concepts are implemented inside the used compilers.

Forbidden forward-edge indirect calls are the result of a vPointer corruption. A vPointer corruption is not a vulnerability but rather a capability which can be the result of a spatial or temporal memory corruption through: (1) bad-casting [25] of C++ objects, (2) buffer overflow in a buffer adjacent to a C++ object or a use-after-free condition [3]. A vPointer corruption can be exploited in several ways. A manipulated vPointer can be exploited by pointing it in any existing or added program vTable entry or into a fake vTable which was added by an attacker. For example in case a vPointer was corrupted then the attacker could hijack the control flow of the program and start a COOP attack [3].

vPointer corruptions are a real security threat which can be exploited if there is a memory corruption (e.g. buffer overflow) which is adjacent to the C++ object or a use-after-free condition. As a consequence each corruption which can reach an object (e.g. bad object casts) is a potential exploit vector for a vPointer corruption. Interestingly to notice in this context is that through: (1) memory layout analysis (through highly configurable compiler tool chains) of source code based locations which are highly prone to memory corruptions such as declarations and uses of buffers, integers or pointer deallocations one can obtain the internal machine code layout representation. (2) analysis of a code

corruption which is adjacent (based on (1)) to a C++ object based on application class hierarchy, the vTble hierarchy and each location in source code where an object is declared and used (e.g., modern compiler tool chains can spill out this information for free), one can derive an analysis which can determine—up to a certain extent—if a memory corruption can influence (is adjacent) to a C++ object.

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

2.4 Real COOP Attack Example

The given example depicted in Figure 2.2 is a proof of concept exploit extracted from [3] and used in order to perform a COOP attack on the Firefox browser. A buffer overflow bug was used in order to call into existing vTable entries by using the a main loop gadget. The attack concludes with opening of an Unix shell. A real-world bug, CVE-2014-3176, was exploited by Crane et al. [13] in order to perform another COOP attack on the Chromium browser. The details of the second attack are far to complex (i.e., involves not properly handled interaction of extensions, IPC, the sync API, and Google V8) and for this reason we briefly present the first documented COOP exploit on a Linux machine.

The C++ class `nsMultiplexInputStream` contains a main loop gadget inside the function `nsMultiplexInputStream::Close(void)` which is performing an indirect calls by dispatching indirect calls on the objects contained in the array. The objects contained in the array during normal execution are of type `nsInputStream` and each of the objects will call the `Close(void)` function in order to close each of the previously opened streams. In order to perform the COOP attack the attacker crafts a C++ program containing a array buffer holding six fake objects. Fake objects can call inside (and outside) the initial class and vTable hierarchies with no constraints. During the attack a buffer is created in order to hold the fake objects. The crafted buffer will be called in stead of the real code in order to call different functions available in the program code. For example the attacker calls a function contained in the class `xpcAccessibleGeneric` which is not in the class hierarchy or vTable hierarchy of the initially intended type of objects used inside the array. Moreover, the header file of this class (`xpcAccessibleGeneric`) is not included in the class `nsMultiplexInputStream`. In total six fake objects are used to call into functions residing in not related class hierarchies with varying num-

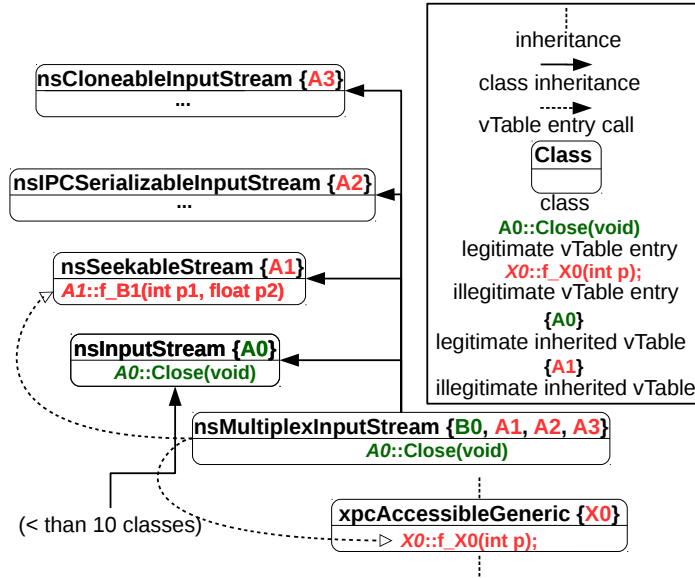


Figure 2.2: Class inheritance hierarchy of the classes involved in the COOP attack against the Firefox browser. Red letters indicate forbidden vTable entries and green letters indicate allowed vTable entries for the given indirect call site contained in the main loop gadget.

ber of parameters and return types. The final goal of this attack is to prepare the program memory such that a Unix shell can be opened at the end of this attack.

This example illustrates why detecting vPointer corruptions is not trivial for real-world applications. As depicted in Figure 2.2 the class `nsInputStream` has 11 classes which inherit directly or indirectly from this class. The classes `nsSeekableStream`, `nsIPCSerializableInputStream` and `nsCloneableInputStream` provide additional inherited vTables which represent illegitimate call targets for the initial `nsInputStream` objects and legitimate call targets for the six fake objects which were added during the attack. Furthermore, declaration and usage of the objects can be wide spread in the source code. This makes detection of the object types (base class), range of vTables (longest vTable inheritance path for a particular call site) and parameter types of the vTable entries (functions) in which it is allowed to call a trivial task for source code (current research work is mostly concerned with performance issues) applications but a hard task in our opinion when one wants to apply similar security policies (e.g. which rely on parameter types of vTable entries) to executables.

3 TypeShield Overview

after the Design and Implementation section is done we pick the most important points of TypeShield design and Implementation and describe them here. The goal of this section is to be an appetizer for the whole design and Implementation section. Which are usually dry (trocken).m

3.1 1. Select Important Point from Design Chapter

3.2 2. Select Important Point from Design Chapter

3.3 3. Select Important Point from Design Chapter

3.4 Adversary Model

We consider a powerful, yet realistic adversary model that is consistent with previous work on code-reuse attacks and mitigations [give COOP reference here]. We rely on several existing and complementary mitigations for comprehensive coverage.

Adversarial Capabilities. 1. System Con

figuration: The adversary is aware of the applied defenses and has access to the source and non-randomized binary of the target application.

2. Vulnerability: The target application suffers from a memory corruption vulnerability that allows the adversary to corrupt memory objects. We assume that the attacker can exploit this aw to read from and write to arbitrary memory addresses.

3. Scripting Environment: The attacker can exploit a scripting environment to process memory disclosure information at run time, adjust the attack payload, and subsequently launch a code-reuse attack.

B. Defensive Requirements. 1. Writable Executable Memory: The target system ensures that memory can be either writable or executable, but not both. This prevents an attacker from either injecting new code or modifying existing executable code.

2. Execute-only Memory: We build on previous systems which enforce execute-only memory pages, i.e., the CPU can fetch instructions but normal read or

write accesses trigger an access violation. See Section 5.2 for further discussion of this component.

3. JIT Protection: We assume mitigations are in place to prevent code injection into the JIT code cache and prevent reuse of JIT compiled code [11, 18, 36]. These protections are orthogonal to Readactor++.

4. Brute-forcing Mitigation: We require that the protected software does not automatically restart after hitting a booby trap which terminates execution. In the browser context, this may be accomplished by displaying a warning message

this section is just an example from the typearmor paper, of course we can replace it with our one but we need to address roughly the same points, namely how TypeShild defends against COOP.

example from coop paper:

In general, code reuse attacks against C++ applications oftentimes start by hijacking a C++ object and its vptr. Attackers achieve this by exploiting a spatial or temporal memory corruption vulnerability such as an overflow in a buffer adjacent to a C++ object or a use-after-free condition. When the application subsequently invokes a virtual function on the hijacked object, the attacker-controlled vptr is dereferenced and a vfptr is loaded from a memory location of the attacker's choice. At this point, the attacker effectively controls the program counter (rip in x64) of the corresponding thread in the target application. Generally for code reuse attacks, controlling the program counter is one of the two basic requirements. The other one is gaining (partial) knowledge on the layout of the target application's address space. Depending on the context, there may exist different techniques to achieve this [8], [28], [44], [48]. For COOP, we assume that the attacker controls a C++ object with a vptr and that she can infer the base address of this object or another auxiliary buffer of sufficient size under her control. Further, she needs to be able to infer the base addresses of a set of C++ modules whose binary layouts are (partly) known to her. For instance, in practice, knowledge on the base address of a single publicly available C++ library in the target address space can be sufficient. These assumptions conform to the attacker settings of most defenses against code reuse attacks. In fact, many of these defenses assume far more powerful adversaries that are, e. g., able to read and write large (or all) parts of an a

3.5 TypeShild: Invariants for Targets and Callsites

this section is just an example from the typearmor paper, of course we can replace it with our one but we need to address roughly the same points, namely how TypeShild defends against COOP.

3.6 TypeShild Impact on COOP

this section is just an example from the typearmor paper, of course we can replace it with our one but we need to address roughly the same points, namely how TypeShild defends against COOP.

4 Design

In the design of Typeshield, we cover various aspects, first of all we introduce the details of the *count* policy - as introduced by [8] and the new *type* policy. Then we need to implement the analysis for calltargets callsites for the the respective policy. For further restriction capabilities we introduce a version of address taken analysis based on [1].

4.1 *count* Policy

What we call the *count* policy is essentially the policy introduced by TypeArmor [8]. The basic idea revolves around classifying calltargets by the number of parameters they provide and callsites by the number of paramters they require. The schema to match those is that we have calltargets requiring parameters and the callsites providing parameters as depicted in figure 4.1.

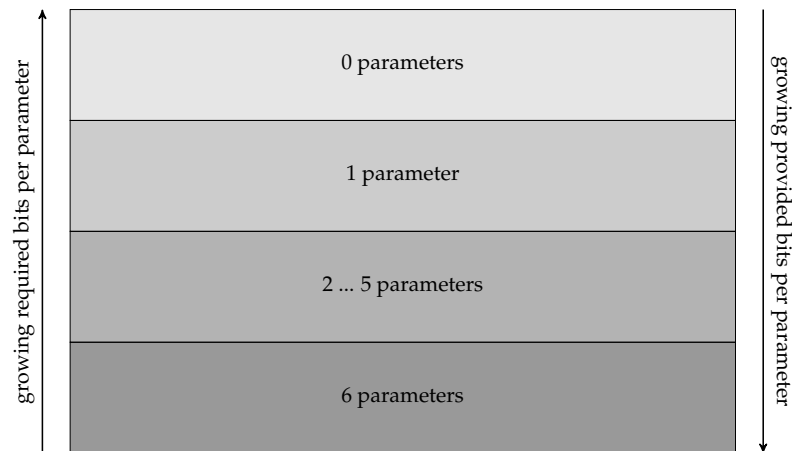


Figure 4.1: *count* policy classification schema

Furthermore, generating 100% precise measurements for such classification with binaries as the only source of information is 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 SystemV ABI - designates as parameter registers. Furthermore, we completely ignore floating point registers or multiinteger registers. The core of the *count* policy is now to allow any callsite cs , which provides c_{cs} parameters, to call any calltarget ct , which requires c_{ct} parameters, iff $c_{ct} \leq c_{cs}$ holds. However, the main problem is that while there is a significant restriction of calltargets for the lower callsites, the restriction capability drops rather rapidly when reaching higher parameter counts, with callsites that use 6 or more parameters being able to call all possible calltargets:

$$\forall cs_1, cs_2. c_{cs_1} \leq c_{cs_2} \implies \|\{ct \in \mathcal{F} | c_{ct} \leq c_{cs_1}\}\| \leq \|\{ct \in \mathcal{F} | c_{ct} \leq c_{cs_2}\}\|$$

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

4.2 *type* Policy

What we call the *type* policy is the idea of not only relying on the parameter count but also on the type of a parameter. However due to complexity reasons, we are restricting ourselves to the general purpose registers, which the SystemV ABI designates as parameter registers. Furthermore we are not inferring the actual type of the data but the wideness of the data stored in the register. The schema again is that we have calltargets requiring wideness and the callsite providing wideness as depicted in figure 4.2.

We are currently interested in x86-64 binaries, the registers we are looking at are 64bit registers that can be accessed in 4 different ways:

- the whole 64bits of the register, meaning a wideness of 64
- the lower 32bits of the register, meaning a wideness of 32
- the lower 16bits of the register, meaning a wideness of 16
- the lower 8bits of the register, meaning a wideness of 8

Four of those registers can also directly access the higher 8bits of the lower 16bits of the register. For our purpose we register this access as a 16bit access. 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. Similar to the *count* policy, we allow overestimation of types

	param 6	param 5	param 4	param 3	param 2	param 1	
	0 bits	0 bits	0 bits	0 bits	0 bits	0 bits	
	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	
	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	
	32 bits	32 bits	32 bits	32 bits	32 bits	32 bits	
	64 bits	64 bits	64 bits	64 bits	64 bits	64 bits	
growing required bits per parameter							growing provided bits per parameter

Figure 4.2: *type* policy schema for callsites and calltargets

in callsites and underestimation of types in calltargets. However, the matching idea is different, because as can we depict in Figure 4.2, the type of a calltarget and a callsite no longer depends solely on its parameter count, each callsite and calltarget has its type from the set of \mathcal{T}^6 , with the following comparison operator:

$$u \leq_{type} v : \Longleftrightarrow \forall_{i=0}^5 u_i \leq v_i, \text{ with } u, v \in \mathcal{T}^6$$

Again we allow any callsite *cs* call any calltarget *ct*, when it fulfills the requirement $ct \leq cs$. The way we represent this is by letting the type for a calltarget parameter progress from 64bit to 0bit - If a calltarget requires a 32bit value in its 1st parameter, it also should accept a 64bit value from its callsite - and similarly we let the type for a callsite progress from 0bit to 64bit - If a callsite provides a 32bit value in its 1st parameter it also provides a 16bit, 8bit and 0bit to a calltarget. Now the advantage of the *type* policy in comparison to the *count* 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. A function that requires 64bit in its first parameter, and 0bit in all other parameters, would have been callable by a callsite providing 8bit in its first and second parameter when using the *count* policy, however in the *type* policy this

is no longer possible.

4.3 Instruction Analysis

Usually data-flow analysis algorithms are based on set of variable or sets of definitions, which both are basically unbounded. However, we are analysing 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)

An instruction $i \in \mathcal{I}$ can non-exclusively perform two kinds of operations on any number of existing registers:

1. Read n bits from the register with $n \in \{64, 32, 16, 8\}$
2. Write n bits to the register with $n \in \{64, 32, 16, 8\}$

Thus we describe the possible change that occurs in one register with the set $S = \{w64, w32, w16, w8, 0\} \times \{r64, r32, r16, r8, 0\}$. Note that 0 signals the absence of either a write or read access and $(0, 0)$ signals the absence of both. Furthermore wn or rn with $n \in \{64, 32, 16, 8\}$ implies all wm or rm with $m \in \{64, 32, 16, 8\}$ and $m < n$ (e.g. $r64$ implies $r32$). Note that we exclude 0, as it means the absence of any access.

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

At last we declare a function, which calculates the change occurring in the processor state, when executing an instruction from \mathcal{I} :

$$decode : \mathcal{I} \mapsto S$$

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

4.4 Calltarget Analysis

For either *count* or *type* policy to work, we need to arrive at an underestimation of the required parameters by any function existing within the targeted binary.

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

4.4.1 Variable Liveness Analysis Theory

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

The book “Data Flow Analysis - Theory and Practice” [?] defines live variable analysis on blocks in the following manner:

$$In_n := (Out_n - Kill_n) \cup Gen_n \quad (4.1a)$$

$$Out_n := \begin{cases} Bl & n \text{ is end block} \\ \bigcup_{s \in succ(n)} In_s & \text{otherwise} \end{cases} \quad (4.1b)$$

Bl is the default state at the end of a path of execution and in our case reaching that state would mean that a variable has never been used (neither written nor read). The set $Kill_n$ describes all variables that are no longer live after the block n , meaning that a variable occurring within this set has been written to. The set Gen_n describes all variables that are alive due to the block n , meaning that a variable occurring within this set has been read before it was written to.

However, we cannot use variable liveness analysis as is, because the analysis is based on potentially unbound variable sets, while we are restricted to a finite number of registers and states. We also require an underestimation of live variables and not an overestimation as provided by standard liveness analysis. Furthermore we have to define how to interpret the changes occurring within one block based on the change caused by its instructions. Considering this, we arrive at algorithm1 to compute the liveness state at the start of a basic block.

This algorithm relies on various functions that can be used to configure its behaviour. We need to define the function *merge_v*, which describes how to compound the state change of the current instruction and the current state, the function *merge_h*, which describes how to merge the states of several paths, the instruction analysis function *analyze_instr*. The function *succ*, which retrieves

Function *analyze*(*block* : *BasicBlock*) : $\mathcal{S}^{\mathcal{L}}$ **is**

```

state = Bl
foreach inst  $\in$  block do
  | state' = analyze_instr(inst)
  | state = merge_v(state, state')
end
states =
blocks = succ(block)
foreach block'  $\in$  blocks do
  | state' = analyse(block')
  | states = states  $\cup$  { state' }
end
state' = merge_h (states)
return merge_v(state, state')
end

```

Algorithm 1: Algorithm to analyse the liveness of a Basic Block

all possible successors of a block won't be implemented by us, because we rely on the DynInst instrumentation framework to achieve this.

$$\textit{merge}_v : \mathcal{S}^{\mathcal{L}} \times \mathcal{S}^{\mathcal{L}} \mapsto \mathcal{S}^{\mathcal{L}} \quad (4.2a)$$

$$\textit{merge}_h : \mathcal{P}(\mathcal{S}^{\mathcal{L}}) \mapsto \mathcal{S}^{\mathcal{L}} \quad (4.2b)$$

$$\textit{analyze_instr} : \mathcal{I} \mapsto \mathcal{S}^{\mathcal{L}} \quad (4.2c)$$

$$\textit{succ} : \mathcal{I} \mapsto \mathcal{P}(\mathcal{I}) \quad (4.2d)$$

As the *analyze_instr* function calculates the effect of an instruction and is the heart of the analyze function. It will also handle non jump and non fallthrough successors, as these are not handled by DynInst in our case. We essentially have four cases that we handle:

1. if the instruction is an indirect call or a direct call but we chose not follow calls, then return a state where all registers are considered written
2. if the instruction is a diect call and we chose to follow calls, then we spawn a new analysis and return its result
3. if the instruction is a constant write (e.g. xor of two registers) then we remove the read portion before we return the decoded state
4. in all other cases we simply return the decoded state

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

4.4.2 Required Parameter Count

To implement the *count* policy, we only need a coarse representation of the state of one register, thus we are interested in the following three different exclusive informations:

1. Was the register written to before its value could be read ?
We represent this with the state W.
2. Was the register read from before its value was overwritten ?
We represent this with the state R.
3. Did neither read nor write access occur for the register ?
We represent this with the state C.

This gives us the following register state $S^{\mathcal{L}} = \{C, R, W\}$ which translates to the register superstate $S^{\mathcal{L}} = (S^{\mathcal{L}})^{16}$. Now, we assume that unless the instructions we are looking at does discard the value it is reading (`xor rax rax` would be such an instruction that we call `const_write`) that reading does preced the writing withing one instruction. Furthermore we are only interested in the first occurrence of a R or W within one path, as following reads or writes do not give us more information. Therefore, we can define our vertical merge function in the following way:

$$merge_v^r(cur, delta) = \begin{cases} delta & cur = C \\ cur & otherwise \end{cases} \quad (4.3)$$

$$merge_v(cur, delta) = (s'_0, \dots, s'_5) \text{ with } s'_j = merge_v^r(cur_j, delta_j) \quad (4.4)$$

Our horizontal merge function is a simple pairwise combination of the given set of states:

$$merge_h(\{s\}) = s \quad (4.5)$$

$$merge_h(\{s\} \cup s') = s \circ merge_h(s') \quad (4.6)$$

We have three viable possibilities for our combination operator \circ , depicted in table 4.1, which all give priority to W :

$\sqcap^{\mathcal{L}}$ is what we call the destructive combination operator, as it returns W on any mismatch

$\sqcap^{\mathcal{L}}$	C	R	W	$\sqcap^{\mathcal{L}}$	C	R	W	$\sqcup^{\mathcal{L}}$	C	R	W
C	C	W	W	C	C	C	W	C	C	R	W
R	W	R	W	R	C	R	W	R	R	R	W
W	W	W	W	W	W	W	W	W	W	W	W

Table 4.1: Different mappings for combining two liveness state values in horizontal matching for the *count* policy

$\sqcap^{\mathcal{L}}$ is what we call the intersection operator, as it returns C, when combining C and R, similar to an intersection

$\sqcup^{\mathcal{L}}$ is what we call the union operator, as it returns R, when combining C and R similar to a union

We apply the liveness analysis for each function with the entry block of the function as start and the return blocks as end and after an analysis run for a function, the index of highest parameter register based on the used callconvention that has the state R is considered to be the number of parameters a function at least requires to be prepared by a callsite.

4.4.3 Required Parameter Wideness

To implement the *type* policy, we need a finer representation of the state of one register, thus we are interested in the following three different not necessarily exclusive informations:

1. Was the register written to before its value could be read ?
We represent this with the state W.
2. How much was read from the register before its value was overwritten?
We represent this with the states $\{ r8, r16, r32, r64 \}$ using R as a placeholder for arbitrary reads.
3. Did neither read nor write access occur for the register ?
We represent this with the state C.

This gives us the following register state $S^{\mathcal{L}} = \{C, r8, r16, r32, r64, W\}$ which translates to the register superstate $\mathcal{S}^{\mathcal{L}} = (S^{\mathcal{L}})^{16}$. Now, we assume that unless the instructions we are looking at does discard the value it is reading (`xor rax, rax` would be such an instruction that we call `const.write`) that reading does precede the writing withing one instruction.

As there could happen 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. We arrive therefore at three possible vertical merge functions:

- The same vertical merge operator as used in the *count* policy, which only gives us the first non *C* state ($merge_v^r$).
- A vertical merge operator that conceptually intersects all read accesses along a path until the first write occurs ($merge_v^i$).
- A vertical merge operator that conceptually calculates the union of all read accesses along a path until the first write occurs ($merge_v^u$).

Our horizontal merge function is a simple pairwise combination of the given set of states:

$$merge_h(\{s\}) = s \quad (4.7)$$

$$merge_h(\{s\} \cup s') = s \circ merge_h(s') \quad (4.8)$$

The results of our experiments with the implementation of calltarget classification gave presented us with essentially one possible candidate that we can base our horizontal merge function on, namely the union operator with an analysis function that follows into direct calls. The basic schema of the merging is depicted in 4.2 and it essentially behaves as if it was the union operator (when both states are set, the higher one is chosen). However, we have to account for W being used as an end marker, which is why we added mapping for RW, which is essentially that.

\cup^L	C	R	W	RW
C	C	R	W	RW
R	R	R^{\cup}	W	$R^{\cup}W$
W	W	W	W	W
RW	RW	$R^{\cup}W$	W	RW

Table 4.2: The union mapping operator for liveness in the *type* policy

4.4.4 Variadic Functions

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

The problem with these functions is that to allow for easier processing of parameters usually all potential variadic parameters are moved into a contiguous

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

Figure 4.3: ASM code of the `make_cmd` function with optimize level O2, which has a variadic parameter list

block of memory, as can be seen in the assembly in figure 4.3. Our analysis interprets that as a read access on all parameters and we arrive at a problematic overestimation.

Our solution to this problem is to find these spurious reads and ignore them. A compiler will implement this type of operation very similar for all cases, thus we can achieve this using the following steps:

1. Look for what we call the xmm-passthrough block, which entirely consist of moving the values of registers `xmm0` to `xmm7` into contiguous memory (in our case basic block [0x422306, 0x42233d [)
2. Look at the predecessor of the xmm-passthrough block, which we call the entry block (in our case basic block [0x4222f0, 0x4222f2 [). Check if the successors of the entry block consist of the xmm-passthrough block and the successor of the xmm-passthrough block, which we call the param-passthrough block (in our case basic block [0x42233d, 0x42235e [).
3. Look at the param-passthrough block and set all instructions that move the value of a parameter register into memory to be ignored (in our case the instructions 0x42233d, 0x422342, 0x422347 and 0x42234c)

4.5 Callsite Analysis

For either *count* or *type* policy to work, we need to arrive at an overestimation of the provided parameters by any indirect callsite existing within the targeted binary. We will employ a modified version of reaching analysis that tracks registers instead of variables to generate the needed overestimation. As our algorithm will be customizable, we look at the required merge functions to implement *count* and *type* policy.

4.5.1 Reaching Definitions Theory

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

The book “Data Flow Analysis - Theory and Practice” [?] defines reaching definition analysis on blocks in the following manner:

$$In_n := \begin{cases} Bl & \text{n is start block} \\ \bigcup_{p \in \text{pred}(n)} Out_p & \text{otherwise} \end{cases} \quad (4.9a)$$

$$Out_n := (In_n - Kill_n) \cup Gen_n \quad (4.9b)$$

Bl is the default state at the start of a path of execution and in our case reaching that state would mean that we do not know whether a value has been provided for the variable and therefore we assume that one has been provided, reaching an overestimation. The set $Kill_n$ describes all definitions that are removed within this block, meaning that the value of a variable has been overwritten. The set Gen_n describes the new definitions that have been provided by the block n , meaning that the value of a variable has been assigned. Considering this, we can assume that $Gen_n \subseteq Kill_n$, as we can always create new definitions, but not simply remove definitions without assigning a new value to the variable.

However, we cannot use reaching definition analysis as is, because the analysis is again based on potentially unbound variable sets, while we are restricted to a finite number of registers and states. This time however the analysis provides us with an overestimation, we however want to get a result as close as possible so we again want to customize merge functions. Furthermore we have to define how to interpret the changes occurring within one block based on the change caused by its instructions. Considering this, we arrive at the following algorithm to compute the liveness state at the start of a basic block:

Function *analyze*(*block* : *BasicBlock*) : \mathcal{S}^R **is**

```

state = Bl
foreach inst  $\in$  reversed(block) do
    state' = analyze_instr(inst)
    state = merge_v(state, state')
end
states =
blocks = pred(block)
foreach block'  $\in$  blocks do
    state' = analyse(block')
    states = states  $\cup$  { state' }
end
state' = merge_h (states)
return merge_v(state, state')

```

end

Algorithm 2: Algorithm to analyse the reaching definitions of a Basic Block

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

$$\text{merge_v} : \mathcal{S}^{\mathcal{R}} \times \mathcal{S}^{\mathcal{R}} \mapsto \mathcal{S}^{\mathcal{L}} \quad (4.10a)$$

$$\text{merge_h} : \mathcal{P}(\mathcal{S}^{\mathcal{R}}) \mapsto \mathcal{S}^{\mathcal{R}} \quad (4.10b)$$

$$\text{analyze_instr} : \mathcal{I} \mapsto \mathcal{S}^{\mathcal{R}} \quad (4.10c)$$

$$\text{pred} : \mathcal{I} \mapsto \mathcal{P}(\mathcal{I}) \quad (4.10d)$$

As the *analyze_instr* function calculates the effect of an instruction and is the heart of the analyze function. It will also handle non jump and non fallthrough 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 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
3. in all other cases we simply return the decoded state

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

4.5.2 Provided Parameter Count

To implement the *count* policy, we only need a coarse representation of the state of one register, thus we are interested in the following three different exclusive informations:

1. Was the register value trashed ?
We represent this with the state T.
2. Was the register written to ?
We represent this with the state S.
3. Was the register neither trashed nor written to ?
We represent this with the state U.

$\sqcap^{\mathcal{R}}$	U	S	T	$\cap^{\mathcal{R}}$	U	S	T	$\cup^{\mathcal{R}}$	U	S	T	$\sqcup^{\mathcal{R}}$	U	S	T
U	U	T	T	U	U	U	T	U	U	S	T	U	U	S	T
S	T	S	T	S	U	S	T	S	S	S	T	S	S	S	S
T	T	T	T	T	T	T	T	T	T	T	T	T	T	S	T

Table 4.3: Different mappings for combining two reaching state values in horizontal matching for the *count* policy

This gives us the following register state $S^{\mathcal{L}} = \{T, S, U\}$ which translates to the register superstate $S^{\mathcal{R}} = (S^{\mathcal{R}})^{16}$. 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, we can define our vertical merge function in the following way:

$$merge_v^r(cur, delta) = \begin{cases} delta & cur = U \\ cur & otherwise \end{cases} \quad (4.11)$$

$$merge_v(cur, delta) = (s'_0, \dots, s'_1 5) \text{ with } s'_j = merge_v^r(cur_j, delta_j) \quad (4.12)$$

Our horizontal merge function is a simple pairwise combination of the given set of states:

$$merge_h(\{s\}) = s \quad (4.13)$$

$$merge_h(\{s\} \cup s') = s \circ merge_h(s') \quad (4.14)$$

We have four viable possibilities for our combination operator \circ , depicted in table 4.3, which all (except one) give priority to *T*:

$\sqcap^{\mathcal{R}}$ is what we call the destructive combination operator, as it returns T on any mismatch

$\cap^{\mathcal{R}}$ is what we call the intersection operator, as it returns U, when combining U and S, similar to an intersection

$\cup^{\mathcal{R}}$ is what we call the union operator, as it returns S, when combining U and S similar to a union

$\sqcup^{\mathcal{R}}$ is what we call the true union operator, as it gives S precedence over everything and returns T or U only when both sides are T or U being more inclusive than a union

4.5.3 Provided Parameter Wideness

To implement the *type* policy, we need a finer representation of the state of one register, thus we are interested in the following three informations:

1. Was the register value trashed ?
We represent this with the state T .
2. Was the register written to and how much ?
We represent this with the states $\{s64, s32, s16, s8\}$ using S as a placeholder for arbitrary writes.
3. Was the register neither trashed nor written to ?
We represent this with the state U .

This gives us the following register state $S^{\mathcal{L}} = \{T, s64, s32, s16, s8, U\}$ which translates to the register superstate $S^{\mathcal{R}} = (S^{\mathcal{L}})^{16}$. Again, we are only interested in the first occurrence of a state that is not U in a path, as following reads or writes do not give us more information.

Therefore we can use the same vertical merge function as for the *count* policy, which is essentially a passthrough until the first non U state.

Our horizontal merge function is again a simple pairwise combination of the given set of states:

$$\text{merge_h}(\{s\}) = s \quad (4.15)$$

$$\text{merge_h}(\{s\} \cup s') = s \circ \text{merge_h}(s') \quad (4.16)$$

However, we have different possibilities regarding the merge operator. Experiments with our implementations for callsite classification in the *count* policy have given us the following results:

- The best candidate to minimize the problematic matches is the union operator without following direct calls
- The best candidate to maximize precision is the intersection operator with following direct calls

We therefore arrive at three viable possibilities for our combination operator \circ , depicted in table 4.4, which all (except one) give priority to T :

$\cap^{\mathcal{R}}$ is what we call the intersection operator, as it returns U , when combining U and S , similar to an intersection furthermore we also calculate the intersection of states when both states are set (the lower of the two is returned)

$\cap^{\mathcal{R}}$	U	S	T	$\sqcap^{\mathcal{R}}$	U	S	T	$\cup^{\mathcal{R}}$	U	S	T
U	U	U	T	U	U	U	T	U	U	S	T
S	U	S^{\cap}	T	S	U	S^{\cup}	T	S	S	S^{\cup}	T
T	T	T	T	T	T	T	T	T	T	T	T

Table 4.4: Different mappings for combining two reaching state values in horizontal matching for the *type* policy

$\sqcap^{\mathcal{R}}$ is what we call the half intersection operator, as it returns U, when combining U and S, similar to an intersection but we calculate the union of states when both states are set (the higher of the two is returned)

$\cup^{\mathcal{R}}$ is what we call the union operator, as it returns S, when combining U and S similar to a union furthermore we calculate the union of states when both states are set (the higher of the two is returned)

Initial experiments with this implementation showed two problems regarding provided wideness detection. Parameter lists with “holes” and address wideness underestimation.

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

Address wideness unterestimation refers to the problem that while in the callsite a constant value of 32bit is written to a register, however the calltarget uses the whole 64bit register. This can occur when pointers are passed from the callsite to the calltarget. Specifically this happens when pointers to memory inside the “.bss”, “.data” or “.rodata” section of the binary are passed. Our solution is to enhance our instruction analysis to watch out for constant writes. In case a 32bit constant value write is detected, we check if the value is an address within the “.bss”, “.data” or “.rodata” section of the binary. If this is

the case, we simply return a write access of 64bits instead of 32bits. (This is not problematic, because we are looking for an overestimation of parameter wideness) It should be noted that the same problem can arise when a constant write causes the value 0 to be written to a 32bit register. We use the same solution and set the wideness to 64bits instead of 32bits.

4.6 Address Taken Analysis

As of now, we use the maximum available set of calltargets - the set of all function entry basic blocks - as input for our algorithm. To restrict the number of calltargets per callsite even further, we explored the possibility of incorporating an address taken analysis into our application. We base our theory on the paper by Zhang and Sekar[1], which introduced various types of taken addresses. An address is considered to be taken, when it is loaded into memory or a register.

4.6.1 Address Taken Targets

Based on the notions of [1], which classified taken addresses into several types of indirect control flow targets, we only chose Code Pointer Constants (CK) and discarded the others:

- Code Pointer Constants (CK) are addresses that are calculated during the compilation of the binary and point within the possible range of addresses in the current module or to instruction boundaries. We are however only interested in addresses that directly point to an entry basic block of a function, as these are the only valid targets for any callsite.
- Computed code pointers (CC) are the result of simple pointer arithmetic, however these are only used for intra procedural jumps. We rely on DynInst to resolve those and only focus on indirect callsites, therefore these are of no interest to us.
- Exception handling addresses (EH) are used to handle exceptions within C++ functions and are modelled as jumps within the function. These are therefore within the normal controlflow that we rely on DynInst to resolve for us.
- Exported function addresses (ES) are essentially functions that point outside of our current module (usually to dynamically linked libraries) and are implemented as jumps, which are of no concern to us, because our analysis is only concerned about the current object.

- Return addresses (RA), which are the addresses next to a call instruction, are also of no interest to us, because we only implement forward control flow integrity.

4.6.2 Binary Analysis

Our approach of identifying taken addresses consists of two steps: First, we iterate over the raw binary content of data sections. Second, we iterate over all functions within the disassembled binary. We rely on DynInst to provide us with the boundaries of the sections inside the binary and in case of shared libraries with the needed translation to current memory addresses:

1. We look at three different data sections of the binary, which could possibly contain taken addresses: the `.data`, `.rodata` and `.dynsym` sections. As [1] proposed, we slide a four byte window over the data within those sections and look for addresses that point to function entry blocks. However, we are looking at x64 binaries therefore we additionally use an eight byte window. In case of shared libraries, we need to let DynInst translate the raw address, we extracted, so we can perform the function check.
2. We specifically look for instructions that load a constant value into a register or memory, and again check whether the address points to the entry block of a function.

5 Implementation

We implemented TypeShield as a module pass for the di-opt environment from patharmor[15], which relies on the DynInst [22] instrumentation framework (we are using version 9.2.0). However, converting to a standalone executable is also possible, as we do not rely on any patharmor feature except for the pass abstraction.

Our module pass relies on Dyninst, to resolve the structure of the binaries that we analyse. The core part of our pass is an instruction analyser, which relies on the DynamoRIO [23] library (version 6.6.1) to decode single instructions and provide access to its information. This analyser is then used to implement basic analyses, especially our version of the reaching and liveness analyses, which can be customised with relative ease, as we allow for arbitrary path merging functions, however we provide the three basic versions (destructive, intersection and union).

Furthermore, we had to patch the DynInst library to allow for local annotation of calltargets with arbitrary information, leveraging its relocation schema, which relies on the BasicBlock abstraction.

Additionally, to measure the quality and performance of our tool, we wrote a pass for the llvm/clang framework version 4.0.0 (trunk 283889) in the x86 target code generation portion, to generate ground truth data. This data is then used to verify the output of our tool for several testtargets, which is done in our python evaluation and test environment.

In total we implemented TypeShield in 5123 lines of C++ code, our clang/llvm-pass in 200 lines of C++ code and our test environment in 2674 lines of python code (all lines of code data is given excluding empty lines and comments).

At this stage of development we are restricted to analysis and instrumentation of x86-64 bit elf binaries using the SystemV call convention, because the DynInst library does not yet support the Windows platform. However, there is currently work being done to allow DynInst to also work with Windows binaries. We restricted ourselves to the SystemV call convention as most C/C++ compilers on linux implement this ABI, however we encapsulated most ABI dependent behaviour, so it should be possible to implement other ABIs with relative ease. Therefore, we deem it possible to implement TypeShield for the Windows platform in the near future, as we do not use any other platform-dependent API's.

6 Evaluation

As we do not have access to the source code of typearmor, we implemented two modes in *TypeShield*. The first mode of our tool is an approximate implementation of what we understand is the COUNT policy implemented by typearmor. The second mode is our implementation of the TYPE policy on top of our implementation of the COUNT policy. In our evaluation of the two modes of *TypeShield*, we are trying to answer the following three questions:

- R1 How precise is *TypeShield* in recovering parameter count and type information for callsites and calltargets from a given binary?
- R2 How effective is *TypeShield* in restricting the possible number of calltargets per callsite?
- R3 What level of security does *TypeShield* offer ?

We evaluated our *TypeShield* by instrumenting various open source applications and analyzing the result. We used the two ftp server applications vsftpd (version 1.1.0) and proftpd (version 1.3.3), the two http server applications postgresql (version 9.0.10) and mysql (5.1.65), the memory cache application memcached (version 1.4.20) and the node.js server application node (version 0.12.5). We chose these applications, which are a subset of the applications also used by the typearmor paper[8] to allow for later comparison.

We setup our environment within a VirtualBox (version 5.0.26r) instance, which runs Kubuntu 16.04 LTS (Linux Kernel version 4.4.0) and has access to 3GB of RAM and 4 of 8 provided hardware threads (Intel i7-4170HQ @ 2.50 GHz).

6.1 Precision of TypeShield

To measure the precision of TypeShield, we need to compare the classification of callsites and calltargets as is given by our tool to some sort of ground truth for our testtargets. We generate this ground truth by compiling our testtargets using a custom compiled clang/llvm compiler (version 4.0.0 trunk 283889) with a MachineFunction pass inside the x86 code generation implementation of llvm. We essentially collect three data points for each callsite/calltarget from our llvm-pass:

1. the point of origination, which is either the name of the calltarget or the name of the function the callsite resides in
2. the return type that is either expected by the callsite or provided by the calltarget
3. the parameter list that is provided by the callsite or expected by the calltarget, which discards the variadic argument list

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

6.1.1 Quality and Applicability of Ground Truth

To assess the applicability of our collected ground truth, we essentially need to assess the structural compatability of our two datasets. First we take a look at the comparability of calltargets, which is quite high throughout optimization levels. Then we take a look at the compatability of callsites, which is qualitatively low in the higher optimization levels, while five of our testtargets start with 0% mismatch in O0, mysql stays throughout all levels at a constant mismatch rate of around 18%, and the others between 2% and 17%.

Calltargets The obvious choice for structural comparison regarding calltargets is their name, as these are simply functions. First, we have to however remove internal functions from our datasets like the `_init` or `_fini` functions, which are of no consequence for us. Furthermore, while C functions can simply be matched by their name as they are unique through the binary, the same cannot be said about the language C++. One of the key differences between C and C++ is function overloading, which allows to define 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 the return type, its namespace and the types of its parameter list. We therefore need to reverse this process, which is called demangling and then compare the fully typed names. The table 6.1 shows three data points regarding calltargets for optimization levels O0, O1, O2 and O3:

1. The number of comparable calltargets that are found in both datasets
2. The number of calltargets that are found by TypeShield but not by our clang/llvm pass, named clang miss

3. The number of calltargets that are found by our clang/llvm pass but not by TypeShield, named tool miss

The problematic column is the clang miss column, as these might indicate problems with TypeShield, however as these numbers are relatively low (below 1%) throughout optimization levels, with only node showing a significant higher value than the rest of around 1.6%. The column labeled tool miss lists higher numbers, however these are of no real concern to us, as our ground truth pass possibly collects more data. The data of all source files used during the compilation of our testtargets is collected, which might generate more than one binary and therefore not necessary all sourcefiles are used for our testtarget.

Considering this, we can safely state that our structural matching between ground truth and TypeShield regarding calltargets is nearly perfect ($>98\%$)

Callsites While our structural matching of calltargets is rather simple, we have not so much luck regarding Callsites. While our tool can provide accurate addressing of callsites within the binary, clang/llvm does not have such capabilities in its intermediate representation. Therefore we assume that the ordering of callsites stays roughly the same within one function and that we exclude all functions, which report a different amount of callsites in both datasets. The table 6.1 shows three data points regarding callsites for optimization levels O0, O1, O2 and O3:

1. The number of comparable callsites that are found in both datasets
2. The number of callsites that are discarded due to mismatch from the dataset of TypeShield, named clang miss
3. The number of callsites that are discarded due to mismatch from the dataset of our clang/llvm pass, named tool miss

Second, we look at callsites and this is more problematic, as clang/llvm does not have a notion of instruction address in its IR, therefore we assume the ordering in a function is the same in both datasets and when the callsite count is not the same for dyninst and clang/padyn, we discard it. There are several reasons for mismatch: One is the tailcall optimization, which means that a call instructions at the end of a function are converted into jump instructions. Another one is callsite merging, which happens when a call to a function exists several times within a function and the compiler can merge the paths to this function. Furthermore we already eliminated multiple compilations of the same sourcefile during one testtarget compilation (this would have skewed the results in the case of memcached).

6 Evaluation

O0						
Target	calltargets			callsites		
	match	clang miss	tool miss	match	clang miss	tool miss
proftpd	1171	3 (0.25%)	19 (1.59%)	174	0 (0.0%)	0 (0.0%)
vsftpd	391	3 (0.76%)	0 (0.0%)	2	0 (0.0%)	0 (0.0%)
lighttpd	354	2 (0.56%)	472 (57.14%)	68	0 (0.0%)	0 (0.0%)
nginx	1098	3 (0.27%)	0 (0.0%)	283	0 (0.0%)	0 (0.0%)
mysqld	13020	16 (0.12%)	5453 (29.51%)	6464	1269 (16.41%)	437 (6.33%)
postgres	9273	4 (0.04%)	2119 (18.6%)	688	1 (0.14%)	0 (0.0%)
memcached	233	1 (0.42%)	87 (27.18%)	47	9 (16.07%)	4 (7.84%)
node	20638	339 (1.61%)	620 (2.91%)	8599	1983 (18.73%)	1894 (18.05%)
O1						
proftpd	1171	3 (0.25%)	19 (1.59%)	70	5 (6.66%)	14 (16.66%)
vsftpd	391	3 (0.76%)	0 (0.0%)	2	0 (0.0%)	0 (0.0%)
lighttpd	352	2 (0.56%)	469 (57.12%)	52	1 (1.88%)	2 (3.7%)
nginx	1097	3 (0.27%)	0 (0.0%)	216	4 (1.81%)	6 (2.7%)
mysqld	12682	16 (0.12%)	5444 (30.03%)	5401	926 (14.63%)	841 (13.47%)
postgres	9265	4 (0.04%)	2118 (18.6%)	443	65 (12.79%)	58 (11.57%)
memcached	232	1 (0.42%)	88 (27.5%)	44	3 (6.38%)	4 (8.33%)
node	20638	339 (1.61%)	620 (2.91%)	8599	1983 (18.73%)	1894 (18.05%)
O2						
proftpd	1015	0 (0.0%)	15 (1.45%)	112	7 (5.88%)	15 (11.81%)
vsftpd	318	0 (0.0%)	0 (0.0%)	14	0 (0.0%)	1 (6.66%)
lighttpd	290	0 (0.0%)	311 (51.74%)	48	7 (12.72%)	8 (14.28%)
nginx	921	0 (0.0%)	0 (0.0%)	234	3 (1.26%)	7 (2.9%)
mysqld	9742	13 (0.13%)	3690 (27.47%)	6671	883 (11.68%)	896 (11.84%)
postgres	6930	1 (0.01%)	1512 (17.91%)	565	71 (11.16%)	66 (10.45%)
memcached	133	0 (0.0%)	91 (40.62%)	47	3 (6.0%)	1 (2.08%)
node	20638	339 (1.61%)	620 (2.91%)	8599	1983 (18.73%)	1894 (18.05%)
O3						
proftpd	1007	0 (0.0%)	14 (1.37%)	136	7 (4.89%)	15 (9.93%)
vsftpd	314	0 (0.0%)	0 (0.0%)	18	0 (0.0%)	1 (5.26%)
lighttpd	289	0 (0.0%)	308 (51.59%)	49	9 (15.51%)	10 (16.94%)
nginx	908	0 (0.0%)	0 (0.0%)	240	3 (1.23%)	7 (2.83%)
mysqld	9703	13 (0.13%)	3665 (27.41%)	6946	953 (12.06%)	959 (12.13%)
postgres	6848	1 (0.01%)	1491 (17.87%)	613	73 (10.64%)	70 (10.24%)
memcached	133	0 (0.0%)	89 (40.09%)	47	3 (6.0%)	1 (2.08%)
node	20638	339 (1.61%)	620 (2.91%)	8599	1983 (18.73%)	1894 (18.05%)

Table 6.1: Table shows the quality of structural matching provided by our automated verify and test environment, regarding callsites and calltargets when compiling with optimization levels O0 through O3. The label clang miss denotes elements not found in the dataset of the clang/llvm pass. The label tool miss denotes elements not found in the dataset of TypeShield

Although normally up to 20% mismatch would not be that much of a problem, however this percentage is only based on the number of callsites per function and we cannot really give any guarantees that the callsites in both datasets are the same. (Although for O0 there is quite a high possibility due to the absence of nearly all optimizations, however `mysqld` and `node` remain problematic).

6.1.2 Precision Calltarget Classification (COUNT)

Experiment Setup To choose the best possible combination operator for the calltarget analysis in implementing the COUNT policy, we generated data for all six possible versions of liveness analysis.

1. destructive combination operator with a *succ* function that does not follow into occurring call functions see Table 6.2 for results
2. destructive combination operator with a *succ* function that follows into occurring call functions see Table 6.2 for results
3. intersection combination operator with a *succ* function that does not follow into occurring call functions see Table 6.2 for results
4. intersection combination operator with a *succ* function that follows into occurring call functions see Table 6.2 for results
5. union combination operator with a *succ* function that does not follow into occurring call functions see Table 6.3 for results
6. union combination operator with a *succ* function that follows into occurring call functions see Table 6.3 for results

For each possible version we measured two data points per testtarget, the number and ratio of perfect classifications and the number and ratio of problematic classifications, which in this case refers to overestimations.

We are going to group our results into two classes based on the optimization level, completely unoptimized (O0) and optimized (O1, O2 and O3) as those two groups are highly different in the values they produce.

Results for unoptimized targets

Results for optimized targets

O0		not following calls		following calls	
Target	#	perfect	problem	perfect	problem
proftpd	1171	1132(96.66%)	30(2.56%)	1064(90.86%)	98(8.36%)
vsftpd	391	388(99.23%)	0(0.0%)	388(99.23%)	0(0.0%)
lighttpd	354	353(99.71%)	1(0.28%)	350(98.87%)	4(1.12%)
nginx	1098	1090(99.27%)	7(0.63%)	1078(98.17%)	19(1.73%)
mysqld	13020	12744(97.88%)	30(0.23%)	12697(97.51%)	77(0.59%)
postgres	9273	9057(97.67%)	14(0.15%)	9037(97.45%)	34(0.36%)
memcached	233	229(98.28%)	2(0.85%)	227(97.42%)	4(1.71%)
node	20638	15124(73.28%)	0(0.0%)	15315(74.2%)	0(0.0%)
O1					
proftpd	1171	1010(86.25%)	0(0.0%)	1027(87.7%)	0(0.0%)
vsftpd	391	288(73.65%)	0(0.0%)	306(78.26%)	0(0.0%)
lighttpd	352	306(86.93%)	0(0.0%)	309(87.78%)	0(0.0%)
nginx	1097	848(77.3%)	1(0.09%)	854(77.84%)	1(0.09%)
mysqld	12682	8469(66.77%)	3(0.02%)	8984(70.84%)	3(0.02%)
postgres	9265	7772(83.88%)	1(0.01%)	7952(85.82%)	1(0.01%)
memcached	232	205(88.36%)	0(0.0%)	206(88.79%)	0(0.0%)
node	20638	15124(73.28%)	0(0.0%)	15315(74.2%)	0(0.0%)
O2					
proftpd	1015	874(86.1%)	0(0.0%)	883(86.99%)	0(0.0%)
vsftpd	318	232(72.95%)	0(0.0%)	248(77.98%)	0(0.0%)
lighttpd	290	251(86.55%)	0(0.0%)	253(87.24%)	0(0.0%)
nginx	921	691(75.02%)	0(0.0%)	695(75.46%)	0(0.0%)
mysqld	9742	6512(66.84%)	1(0.01%)	6567(67.4%)	1(0.01%)
postgres	6930	5752(83.0%)	0(0.0%)	5878(84.81%)	0(0.0%)
memcached	133	117(87.96%)	0(0.0%)	117(87.96%)	0(0.0%)
node	20638	15124(73.28%)	0(0.0%)	15315(74.2%)	0(0.0%)
O3					
proftpd	1007	871(86.49%)	0(0.0%)	879(87.28%)	0(0.0%)
vsftpd	314	230(73.24%)	0(0.0%)	246(78.34%)	0(0.0%)
lighttpd	289	253(87.54%)	0(0.0%)	253(87.54%)	0(0.0%)
nginx	908	677(74.55%)	0(0.0%)	681(75.0%)	0(0.0%)
mysqld	9703	6472(66.7%)	0(0.0%)	6526(67.25%)	0(0.0%)
postgres	6848	5687(83.04%)	0(0.0%)	5802(84.72%)	0(0.0%)
memcached	133	117(87.96%)	0(0.0%)	117(87.96%)	0(0.0%)
node	20638	15124(73.28%)	0(0.0%)	15315(74.2%)	0(0.0%)

Table 6.2: The results for calltarget analysis using the destructive/intersection combination operator for the COUNT policy throughout different optimizations

O0 Target	#	not following calls		following calls	
		perfect	problem	perfect	problem
proftpd	1171	1132(96.66%)	30(2.56%)	803(68.57%)	359(30.65%)
vsftpd	391	388(99.23%)	0(0.0%)	388(99.23%)	0(0.0%)
lighttpd	354	353(99.71%)	1(0.28%)	349(98.58%)	5(1.41%)
nginx	1098	1090(99.27%)	7(0.63%)	1004(91.43%)	93(8.46%)
mysqld	13020	12744(97.88%)	30(0.23%)	11193(85.96%)	1581(12.14%)
postgres	9273	9057(97.67%)	14(0.15%)	9032(97.4%)	39(0.42%)
memcached	233	229(98.28%)	2(0.85%)	223(95.7%)	8(3.43%)
node	20638	15626(75.71%)	0(0.0%)	15863(76.86%)	0(0.0%)
O1					
proftpd	1171	1020(87.1%)	0(0.0%)	1066(91.03%)	0(0.0%)
vsftpd	391	289(73.91%)	0(0.0%)	307(78.51%)	0(0.0%)
lighttpd	352	320(90.9%)	0(0.0%)	323(91.76%)	0(0.0%)
nginx	1097	880(80.21%)	1(0.09%)	888(80.94%)	1(0.09%)
mysqld	12682	8626(68.01%)	4(0.03%)	9162(72.24%)	5(0.03%)
postgres	9265	7921(85.49%)	1(0.01%)	8115(87.58%)	1(0.01%)
memcached	232	209(90.08%)	0(0.0%)	210(90.51%)	0(0.0%)
node	20638	15626(75.71%)	0(0.0%)	15863(76.86%)	0(0.0%)
O2					
proftpd	1015	880(86.69%)	0(0.0%)	920(90.64%)	0(0.0%)
vsftpd	318	233(73.27%)	0(0.0%)	249(78.3%)	0(0.0%)
lighttpd	290	266(91.72%)	0(0.0%)	268(92.41%)	0(0.0%)
nginx	921	720(78.17%)	0(0.0%)	724(78.61%)	0(0.0%)
mysqld	9742	6684(68.61%)	1(0.01%)	6759(69.38%)	1(0.01%)
postgres	6930	5865(84.63%)	0(0.0%)	6001(86.59%)	0(0.0%)
memcached	133	117(87.96%)	0(0.0%)	117(87.96%)	0(0.0%)
node	20638	15626(75.71%)	0(0.0%)	15863(76.86%)	0(0.0%)
O3					
proftpd	1007	877(87.09%)	0(0.0%)	913(90.66%)	0(0.0%)
vsftpd	314	231(73.56%)	0(0.0%)	247(78.66%)	0(0.0%)
lighttpd	289	268(92.73%)	0(0.0%)	268(92.73%)	0(0.0%)
nginx	908	708(77.97%)	0(0.0%)	712(78.41%)	0(0.0%)
mysqld	9703	6655(68.58%)	0(0.0%)	6730(69.35%)	0(0.0%)
postgres	6848	5803(84.74%)	0(0.0%)	5926(86.53%)	0(0.0%)
memcached	133	117(87.96%)	0(0.0%)	117(87.96%)	0(0.0%)
node	20638	15626(75.71%)	0(0.0%)	15863(76.86%)	0(0.0%)

Table 6.3: The results for calltarget analysis using the union combination operator for the COUNT policy throughout different optimizations

6.1.3 Precision Callsite Classification (COUNT)

Without interprocedural Analysis

Experiment Setup To choose the best possible combination operator for the callsite analysis in implementing the COUNT policy without a backward interprocedural analysis, we generated data for all eight possible versions of reaching definition analysis.

1. destructive combination operator with a *succ* function that does not follow into occurring call functions see Table 6.4 for results
2. destructive combination operator with a *succ* function that follows into occurring call functions see Table 6.4 for results
3. intersection combination operator with a *succ* function that does not follow into occurring call functions see Table 6.4 for results
4. intersection combination operator with a *succ* function that follows into occurring call functions see Table 6.4 for results
5. union combination operator with a *succ* function that does not follow into occurring call functions see Table 6.5 for results
6. union combination operator with a *succ* function that follows into occurring call functions see Table 6.5 for results
7. union combination operator with a *succ* function that does not follow into occurring call functions see Table 6.6 for results
8. union combination operator with a *succ* function that follows into occurring call functions see Table 6.6 for results

For each possible version we measured two data points per testtarget, the number and ratio of perfect classifications and the number and ratio of problematic classifications, which in this case refers to underestimations.

We are going to group our results into two classes based on the optimization level, completely unoptimized (O0) and optimized (O1, O2 and O3) as those two groups are highly different in the values they produce.

Results for unoptimized targets

Results for optimized targets

O0 Target	#	not following calls		following calls	
		perfect	problem	perfect	problem
proftpd	174	106(60.91%)	0(0.0%)	106(60.91%)	0(0.0%)
vsftpd	2	1(50.0%)	0(0.0%)	1(50.0%)	0(0.0%)
lighttpd	68	16(23.52%)	0(0.0%)	16(23.52%)	0(0.0%)
nginx	283	129(45.58%)	0(0.0%)	129(45.58%)	0(0.0%)
mysqld	6464	746(11.54%)	77(1.19%)	746(11.54%)	77(1.19%)
postgres	688	247(35.9%)	10(1.45%)	247(35.9%)	10(1.45%)
memcached	47	1(2.12%)	0(0.0%)	1(2.12%)	0(0.0%)
node	8599	5886(68.44%)	489(5.68%)	5886(68.44%)	489(5.68%)
O1					
proftpd	70	57(81.42%)	7(10.0%)	57(81.42%)	7(10.0%)
vsftpd	2	2(100.0%)	0(0.0%)	2(100.0%)	0(0.0%)
lighttpd	52	37(71.15%)	9(17.3%)	37(71.15%)	9(17.3%)
nginx	216	172(79.62%)	24(11.11%)	172(79.62%)	24(11.11%)
mysqld	5401	4038(74.76%)	362(6.7%)	4038(74.76%)	362(6.7%)
postgres	443	375(84.65%)	27(6.09%)	375(84.65%)	27(6.09%)
memcached	44	39(88.63%)	0(0.0%)	39(88.63%)	0(0.0%)
node	8599	5886(68.44%)	489(5.68%)	5886(68.44%)	489(5.68%)
O2					
proftpd	112	100(89.28%)	4(3.57%)	100(89.28%)	4(3.57%)
vsftpd	14	14(100.0%)	0(0.0%)	14(100.0%)	0(0.0%)
lighttpd	48	34(70.83%)	8(16.66%)	34(70.83%)	8(16.66%)
nginx	234	181(77.35%)	27(11.53%)	181(77.35%)	27(11.53%)
mysqld	6671	4758(71.32%)	673(10.08%)	4758(71.32%)	673(10.08%)
postgres	565	484(85.66%)	30(5.3%)	484(85.66%)	30(5.3%)
memcached	47	39(82.97%)	0(0.0%)	39(82.97%)	0(0.0%)
node	8599	5886(68.44%)	489(5.68%)	5886(68.44%)	489(5.68%)
O3					
proftpd	136	122(89.7%)	6(4.41%)	122(89.7%)	6(4.41%)
vsftpd	18	18(100.0%)	0(0.0%)	18(100.0%)	0(0.0%)
lighttpd	49	35(71.42%)	8(16.32%)	35(71.42%)	8(16.32%)
nginx	240	186(77.5%)	27(11.25%)	186(77.5%)	27(11.25%)
mysqld	6946	4956(71.35%)	727(10.46%)	4956(71.35%)	727(10.46%)
postgres	613	535(87.27%)	29(4.73%)	535(87.27%)	29(4.73%)
memcached	47	39(82.97%)	0(0.0%)	39(82.97%)	0(0.0%)
node	8599	5886(68.44%)	489(5.68%)	5886(68.44%)	489(5.68%)

Table 6.4: The results for callsite analysis using the destructive/intersection combination operator for the COUNT without interprocedural analysis policy throughout different optimizations

O0		not following calls		following calls	
Target	#	perfect	problem	perfect	problem
proftpd	174	63(36.2%)	0(0.0%)	65(37.35%)	0(0.0%)
vsftpd	2	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)
lighttpd	68	1(1.47%)	0(0.0%)	1(1.47%)	0(0.0%)
nginx	283	23(8.12%)	0(0.0%)	26(9.18%)	0(0.0%)
mysqld	6464	466(7.2%)	77(1.19%)	352(5.44%)	77(1.19%)
postgres	688	76(11.04%)	10(1.45%)	100(14.53%)	10(1.45%)
memcached	47	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)
node	8599	2187(25.43%)	102(1.18%)	2277(26.47%)	140(1.62%)
O1					
proftpd	70	25(35.71%)	0(0.0%)	7(10.0%)	1(1.42%)
vsftpd	2	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)
lighttpd	52	2(3.84%)	0(0.0%)	0(0.0%)	3(5.76%)
nginx	216	60(27.77%)	4(1.85%)	52(24.07%)	6(2.77%)
mysqld	5401	1490(27.58%)	70(1.29%)	1546(28.62%)	106(1.96%)
postgres	443	148(33.4%)	0(0.0%)	132(29.79%)	4(0.9%)
memcached	44	30(68.18%)	0(0.0%)	28(63.63%)	0(0.0%)
node	8599	2187(25.43%)	102(1.18%)	2277(26.47%)	144(1.67%)
O2					
proftpd	112	56(50.0%)	0(0.0%)	23(20.53%)	1(0.89%)
vsftpd	14	4(28.57%)	0(0.0%)	1(7.14%)	0(0.0%)
lighttpd	48	2(4.16%)	0(0.0%)	0(0.0%)	4(8.33%)
nginx	234	66(28.2%)	4(1.7%)	56(23.93%)	5(2.13%)
mysqld	6671	1921(28.79%)	112(1.67%)	2044(30.64%)	199(2.98%)
postgres	565	179(31.68%)	0(0.0%)	165(29.2%)	4(0.7%)
memcached	47	30(63.82%)	0(0.0%)	29(61.7%)	0(0.0%)
node	8599	2181(25.36%)	101(1.17%)	2280(26.51%)	143(1.66%)
O3					
proftpd	136	74(54.41%)	0(0.0%)	32(23.52%)	1(0.73%)
vsftpd	18	6(33.33%)	0(0.0%)	5(27.77%)	0(0.0%)
lighttpd	49	2(4.08%)	0(0.0%)	0(0.0%)	4(8.16%)
nginx	240	69(28.75%)	4(1.66%)	62(25.83%)	6(2.5%)
mysqld	6946	2037(29.32%)	110(1.58%)	2173(31.28%)	198(2.85%)
postgres	613	207(33.76%)	0(0.0%)	198(32.3%)	4(0.65%)
memcached	47	30(63.82%)	0(0.0%)	28(59.57%)	0(0.0%)
node	8599	2187(25.43%)	102(1.18%)	2279(26.5%)	141(1.63%)

Table 6.5: The results for callsite analysis using the union combination operator for the COUNT without interprocedural analysis policy throughout different optimizations

O0		not following calls		following calls	
Target	#	perfect	problem	perfect	problem
proftpd	174	61(35.05%)	0(0.0%)	65(37.35%)	0(0.0%)
vsftpd	2	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)
lighttpd	68	0(0.0%)	0(0.0%)	1(1.47%)	0(0.0%)
nginx	283	30(10.6%)	0(0.0%)	24(8.48%)	0(0.0%)
mysqld	6464	436(6.74%)	77(1.19%)	326(5.04%)	77(1.19%)
postgres	688	91(13.22%)	10(1.45%)	88(12.79%)	10(1.45%)
memcached	47	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)
node	8599	2338(27.18%)	132(1.53%)	2145(24.94%)	139(1.61%)
O1					
proftpd	70	22(31.42%)	0(0.0%)	6(8.57%)	1(1.42%)
vsftpd	2	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)
lighttpd	52	1(1.92%)	0(0.0%)	0(0.0%)	3(5.76%)
nginx	216	49(22.68%)	4(1.85%)	45(20.83%)	6(2.77%)
mysqld	5401	1403(25.97%)	70(1.29%)	1434(26.55%)	106(1.96%)
postgres	443	143(32.27%)	0(0.0%)	113(25.5%)	4(0.9%)
memcached	44	29(65.9%)	0(0.0%)	26(59.09%)	0(0.0%)
node	8599	2121(24.66%)	101(1.17%)	2143(24.92%)	143(1.66%)
O2					
proftpd	112	53(47.32%)	0(0.0%)	20(17.85%)	1(0.89%)
vsftpd	14	4(28.57%)	0(0.0%)	1(7.14%)	0(0.0%)
lighttpd	48	1(2.08%)	0(0.0%)	0(0.0%)	4(8.33%)
nginx	234	54(23.07%)	4(1.7%)	49(20.94%)	5(2.13%)
mysqld	6671	1772(26.56%)	112(1.67%)	1934(28.99%)	199(2.98%)
postgres	565	174(30.79%)	0(0.0%)	141(24.95%)	4(0.7%)
memcached	47	29(61.7%)	0(0.0%)	26(55.31%)	0(0.0%)
node	8599	2121(24.66%)	101(1.17%)	2154(25.04%)	143(1.66%)
O3					
proftpd	136	71(52.2%)	0(0.0%)	31(22.79%)	1(0.73%)
vsftpd	18	6(33.33%)	0(0.0%)	3(16.66%)	0(0.0%)
lighttpd	49	1(2.04%)	0(0.0%)	0(0.0%)	4(8.16%)
nginx	240	56(23.33%)	4(1.66%)	55(22.91%)	6(2.5%)
mysqld	6946	1874(26.97%)	110(1.58%)	2066(29.74%)	198(2.85%)
postgres	613	202(32.95%)	0(0.0%)	173(28.22%)	4(0.65%)
memcached	47	29(61.7%)	0(0.0%)	26(55.31%)	0(0.0%)
node	8599	2127(24.73%)	102(1.18%)	2156(25.07%)	141(1.63%)

Table 6.6: The results for callsite analysis using the true union combination operator for the COUNT without interprocedural analysis policy throughout different optimizations

With interprocedural Analysis

Experiment Setup To choose the best possible combination operator for the callsite analysis in implementing the COUNT policy with a backward interprocedural analysis, we generated data for all eight possible versions of reaching definition analysis.

1. destructive combination operator with a *succ* function that does not follow into occurring call functions see Table 6.7 for results
2. destructive combination operator with a *succ* function that follows into occurring call functions see Table 6.7 for results
3. intersection combination operator with a *succ* function that does not follow into occurring call functions see Table 6.7 for results
4. intersection combination operator with a *succ* function that follows into occurring call functions see Table 6.7 for results
5. union combination operator with a *succ* function that does not follow into occurring call functions see Table 6.8 for results
6. union combination operator with a *succ* function that follows into occurring call functions see Table 6.8 for results
7. union combination operator with a *succ* function that does not follow into occurring call functions see Table 6.9 for results
8. union combination operator with a *succ* function that follows into occurring call functions see Table 6.9 for results

For each possible version we measured two data points per testtarget, the number and ratio of perfect classifications and the number and ratio of problematic classifications, which in this case refers to underestimations.

We are going to group our results into two classes based on the optimization level, completely unoptimized (O0) and optimized (O1, O2 and O3) as those two groups are highly different in the values they produce.

Results for unoptimized targets**Results for optimized targets**

O0 Target	#	not following calls		following calls	
		perfect	problem	perfect	problem
proftpd	174	106(60.91%)	0(0.0%)	106(60.91%)	0(0.0%)
vsftpd	2	1(50.0%)	0(0.0%)	1(50.0%)	0(0.0%)
lighttpd	68	16(23.52%)	0(0.0%)	16(23.52%)	0(0.0%)
nginx	283	130(45.93%)	0(0.0%)	130(45.93%)	0(0.0%)
mysqld	6464	780(12.06%)	77(1.19%)	780(12.06%)	77(1.19%)
postgres	688	260(37.79%)	10(1.45%)	260(37.79%)	10(1.45%)
memcached	47	2(4.25%)	0(0.0%)	2(4.25%)	0(0.0%)
node	8599	5947(69.15%)	493(5.73%)	5947(69.15%)	493(5.73%)
O1					
proftpd	70	57(81.42%)	7(10.0%)	57(81.42%)	7(10.0%)
vsftpd	2	2(100.0%)	0(0.0%)	2(100.0%)	0(0.0%)
lighttpd	52	37(71.15%)	9(17.3%)	37(71.15%)	9(17.3%)
nginx	216	172(79.62%)	25(11.57%)	172(79.62%)	25(11.57%)
mysqld	5401	4075(75.44%)	366(6.77%)	4075(75.44%)	366(6.77%)
postgres	443	380(85.77%)	27(6.09%)	380(85.77%)	27(6.09%)
memcached	44	40(90.9%)	0(0.0%)	40(90.9%)	0(0.0%)
node	8599	5947(69.15%)	493(5.73%)	5947(69.15%)	493(5.73%)
O2					
proftpd	112	100(89.28%)	4(3.57%)	100(89.28%)	4(3.57%)
vsftpd	14	14(100.0%)	0(0.0%)	14(100.0%)	0(0.0%)
lighttpd	48	34(70.83%)	8(16.66%)	34(70.83%)	8(16.66%)
nginx	234	181(77.35%)	28(11.96%)	181(77.35%)	28(11.96%)
mysqld	6671	4789(71.78%)	673(10.08%)	4789(71.78%)	673(10.08%)
postgres	565	489(86.54%)	30(5.3%)	489(86.54%)	30(5.3%)
memcached	47	44(93.61%)	0(0.0%)	44(93.61%)	0(0.0%)
node	8599	5947(69.15%)	493(5.73%)	5947(69.15%)	493(5.73%)
O3					
proftpd	136	122(89.7%)	6(4.41%)	122(89.7%)	6(4.41%)
vsftpd	18	18(100.0%)	0(0.0%)	18(100.0%)	0(0.0%)
lighttpd	49	35(71.42%)	8(16.32%)	35(71.42%)	8(16.32%)
nginx	240	186(77.5%)	28(11.66%)	186(77.5%)	28(11.66%)
mysqld	6946	4985(71.76%)	727(10.46%)	4985(71.76%)	727(10.46%)
postgres	613	540(88.09%)	29(4.73%)	540(88.09%)	29(4.73%)
memcached	47	44(93.61%)	0(0.0%)	44(93.61%)	0(0.0%)
node	8599	5947(69.15%)	493(5.73%)	5947(69.15%)	493(5.73%)

Table 6.7: The results for callsite analysis using the destructive/intersection combination operator for the COUNT with interprocedural analysis policy throughout different optimizations

O0		not following calls		following calls	
Target	#	perfect	problem	perfect	problem
proftpd	174	63(36.2%)	0(0.0%)	83(47.7%)	0(0.0%)
vsftpd	2	1(50.0%)	0(0.0%)	1(50.0%)	0(0.0%)
lighttpd	68	1(1.47%)	0(0.0%)	7(10.29%)	0(0.0%)
nginx	283	32(11.3%)	0(0.0%)	78(27.56%)	0(0.0%)
mysqld	6464	480(7.42%)	77(1.19%)	490(7.58%)	77(1.19%)
postgres	688	102(14.82%)	10(1.45%)	157(22.81%)	10(1.45%)
memcached	47	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)
node	8599	2424(28.18%)	132(1.53%)	3455(40.17%)	261(3.03%)
O1					
proftpd	70	25(35.71%)	0(0.0%)	32(45.71%)	4(5.71%)
vsftpd	2	0(0.0%)	0(0.0%)	1(50.0%)	0(0.0%)
lighttpd	52	13(25.0%)	0(0.0%)	9(17.3%)	7(13.46%)
nginx	216	76(35.18%)	4(1.85%)	123(56.94%)	12(5.55%)
mysqld	5401	1570(29.06%)	79(1.46%)	2425(44.89%)	208(3.85%)
postgres	443	159(35.89%)	1(0.22%)	231(52.14%)	7(1.58%)
memcached	44	30(68.18%)	0(0.0%)	36(81.81%)	0(0.0%)
node	8599	2410(28.02%)	129(1.5%)	3459(40.22%)	269(3.12%)
O2					
proftpd	112	56(50.0%)	0(0.0%)	53(47.32%)	2(1.78%)
vsftpd	14	4(28.57%)	0(0.0%)	8(57.14%)	0(0.0%)
lighttpd	48	14(29.16%)	0(0.0%)	13(27.08%)	6(12.5%)
nginx	234	81(34.61%)	6(2.56%)	141(60.25%)	13(5.55%)
mysqld	6671	2079(31.16%)	141(2.11%)	2775(41.59%)	353(5.29%)
postgres	565	189(33.45%)	0(0.0%)	315(55.75%)	7(1.23%)
memcached	47	30(63.82%)	0(0.0%)	38(80.85%)	0(0.0%)
node	8599	2418(28.11%)	137(1.59%)	3473(40.38%)	256(2.97%)
O3					
proftpd	136	74(54.41%)	0(0.0%)	70(51.47%)	2(1.47%)
vsftpd	18	6(33.33%)	0(0.0%)	14(77.77%)	0(0.0%)
lighttpd	49	14(28.57%)	0(0.0%)	12(24.48%)	6(12.24%)
nginx	240	90(37.5%)	5(2.08%)	134(55.83%)	15(6.25%)
mysqld	6946	2202(31.7%)	143(2.05%)	2939(42.31%)	383(5.51%)
postgres	613	217(35.39%)	0(0.0%)	353(57.58%)	9(1.46%)
memcached	47	30(63.82%)	0(0.0%)	38(80.85%)	0(0.0%)
node	8599	2411(28.03%)	132(1.53%)	3454(40.16%)	256(2.97%)

Table 6.8: The results for callsite analysis using the union combination operator for the COUNT with interprocedural analysis policy throughout different optimizations

O0		not following calls		following calls	
Target	#	perfect	problem	perfect	problem
proftpd	174	61(35.05%)	0(0.0%)	83(47.7%)	0(0.0%)
vsftpd	2	0(0.0%)	0(0.0%)	1(50.0%)	0(0.0%)
lighttpd	68	0(0.0%)	0(0.0%)	7(10.29%)	0(0.0%)
nginx	283	30(10.6%)	0(0.0%)	78(27.56%)	0(0.0%)
mysqld	6464	436(6.74%)	77(1.19%)	490(7.58%)	77(1.19%)
postgres	688	91(13.22%)	10(1.45%)	157(22.81%)	10(1.45%)
memcached	47	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)
node	8599	2338(27.18%)	132(1.53%)	3448(40.09%)	264(3.07%)
O1					
proftpd	70	22(31.42%)	0(0.0%)	32(45.71%)	4(5.71%)
vsftpd	2	0(0.0%)	0(0.0%)	1(50.0%)	0(0.0%)
lighttpd	52	1(1.92%)	0(0.0%)	9(17.3%)	7(13.46%)
nginx	216	62(28.7%)	4(1.85%)	123(56.94%)	12(5.55%)
mysqld	5401	1456(26.95%)	79(1.46%)	2425(44.89%)	208(3.85%)
postgres	443	149(33.63%)	1(0.22%)	231(52.14%)	7(1.58%)
memcached	44	29(65.9%)	0(0.0%)	36(81.81%)	0(0.0%)
node	8599	2323(27.01%)	129(1.5%)	3463(40.27%)	270(3.13%)
O2					
proftpd	112	53(47.32%)	0(0.0%)	53(47.32%)	2(1.78%)
vsftpd	14	4(28.57%)	0(0.0%)	8(57.14%)	0(0.0%)
lighttpd	48	2(4.16%)	0(0.0%)	13(27.08%)	6(12.5%)
nginx	234	71(30.34%)	6(2.56%)	141(60.25%)	13(5.55%)
mysqld	6671	1907(28.58%)	141(2.11%)	2775(41.59%)	353(5.29%)
postgres	565	183(32.38%)	0(0.0%)	315(55.75%)	7(1.23%)
memcached	47	29(61.7%)	0(0.0%)	38(80.85%)	0(0.0%)
node	8599	2330(27.09%)	137(1.59%)	3457(40.2%)	256(2.97%)
O3					
proftpd	136	71(52.2%)	0(0.0%)	70(51.47%)	2(1.47%)
vsftpd	18	6(33.33%)	0(0.0%)	14(77.77%)	0(0.0%)
lighttpd	49	2(4.08%)	0(0.0%)	12(24.48%)	6(12.24%)
nginx	240	78(32.5%)	5(2.08%)	134(55.83%)	15(6.25%)
mysqld	6946	2013(28.98%)	143(2.05%)	2939(42.31%)	383(5.51%)
postgres	613	211(34.42%)	0(0.0%)	353(57.58%)	9(1.46%)
memcached	47	29(61.7%)	0(0.0%)	38(80.85%)	0(0.0%)
node	8599	2324(27.02%)	132(1.53%)	3460(40.23%)	259(3.01%)

Table 6.9: The results for callsite analysis using the true union combination operator for the COUNT with interprocedural analysis policy throughout different optimizations

6.1.4 Precision Calltarget Classification (TYPE)

Experiment Setup To choose the best possible combination operator for the calltarget analysis in implementing the TYPE policy, we conducted four experiments based on the data of the Precision Calltarget Classification(COUNT) experiment:

- exp1 union combination operator with a *succ* function that does not follow into occurring call functions and a vertical merge that only accepts the first change see Table 6.10 for results
- exp2 union combination operator with a *succ* function that does not follow into occurring call functions and a vertical merge that unions all reads until the first write see Table 6.10 for results
- exp3 union combination operator with a *succ* function that does not follow into occurring call functions and a vertical merge that intersects all reads until the first write see Table 6.11 for results
- exp4 full union combination operator with a *succ* function that does not follow into occurring call functions and a vertical merge that unions all reads until the first write see Table 6.11 for results

For each possible version we measured two data points per testtarget, the number and ratio of perfect classifications and the number and ratio of problematic classifications, which in this case refers to overestimations.

We are going to group our results into two classes based on the optimization level, completely unoptimized (O0) and optimized (O1, O2 and O3) as those two groups are highly different in the values they produce.

Results for unoptimized targets

Results for optimized targets

O0		exp1		exp2	
Target	#	perfect	problem	perfect	problem
proftpd	1171	803(68.57%)	359(30.65%)	803(68.57%)	359(30.65%)
vsftpd	391	388(99.23%)	0(0.0%)	388(99.23%)	0(0.0%)
lighttpd	354	349(98.58%)	5(1.41%)	349(98.58%)	5(1.41%)
nginx	1098	1004(91.43%)	93(8.46%)	1004(91.43%)	93(8.46%)
mysqld	13020	11193(85.96%)	1581(12.14%)	11193(85.96%)	1581(12.14%)
postgres	9273	9032(97.4%)	39(0.42%)	9031(97.39%)	39(0.42%)
memcached	233	223(95.7%)	8(3.43%)	223(95.7%)	8(3.43%)
node	20638	13673(66.25%)	479(2.32%)	13671(66.24%)	438(2.12%)
O1					
proftpd	1171	996(85.05%)	11(0.93%)	994(84.88%)	11(0.93%)
vsftpd	391	279(71.35%)	4(1.02%)	279(71.35%)	4(1.02%)
lighttpd	352	295(83.8%)	4(1.13%)	297(84.37%)	1(0.28%)
nginx	1097	758(69.09%)	1(0.09%)	754(68.73%)	1(0.09%)
mysqld	12682	7947(62.66%)	447(3.52%)	7946(62.65%)	428(3.37%)
postgres	9265	7230(78.03%)	701(7.56%)	7236(78.1%)	678(7.31%)
memcached	232	191(82.32%)	15(6.46%)	194(83.62%)	12(5.17%)
node	20638	13673(66.25%)	479(2.32%)	13671(66.24%)	438(2.12%)
O2					
proftpd	1015	855(84.23%)	11(1.08%)	852(83.94%)	11(1.08%)
vsftpd	318	230(72.32%)	3(0.94%)	230(72.32%)	3(0.94%)
lighttpd	290	239(82.41%)	6(2.06%)	240(82.75%)	3(1.03%)
nginx	921	599(65.03%)	0(0.0%)	594(64.49%)	0(0.0%)
mysqld	9742	5712(58.63%)	327(3.35%)	5705(58.56%)	316(3.24%)
postgres	6930	5291(76.34%)	585(8.44%)	5291(76.34%)	572(8.25%)
memcached	133	104(78.19%)	10(7.51%)	103(77.44%)	10(7.51%)
node	20638	13673(66.25%)	479(2.32%)	13671(66.24%)	438(2.12%)
O3					
proftpd	1007	848(84.21%)	11(1.09%)	845(83.91%)	11(1.09%)
vsftpd	314	228(72.61%)	3(0.95%)	228(72.61%)	3(0.95%)
lighttpd	289	239(82.69%)	6(2.07%)	240(83.04%)	3(1.03%)
nginx	908	586(64.53%)	0(0.0%)	581(63.98%)	0(0.0%)
mysqld	9703	5695(58.69%)	315(3.24%)	5690(58.64%)	302(3.11%)
postgres	6848	5231(76.38%)	571(8.33%)	5231(76.38%)	556(8.11%)
memcached	133	104(78.19%)	10(7.51%)	103(77.44%)	10(7.51%)
node	20638	13673(66.25%)	479(2.32%)	13671(66.24%)	438(2.12%)

Table 6.10: The results for calltarget analysis for exp1 and exp2 for the TYPE policy throughout different optimizations

O0		exp3		exp4	
Target	#	perfect	problem	perfect	problem
proftpd	1171	802(68.48%)	363(30.99%)	1132(96.66%)	30(2.56%)
vsftpd	391	388(99.23%)	0(0.0%)	388(99.23%)	0(0.0%)
lighttpd	354	349(98.58%)	5(1.41%)	353(99.71%)	1(0.28%)
nginx	1098	1004(91.43%)	93(8.46%)	1090(99.27%)	7(0.63%)
mysqld	13020	11178(85.85%)	1606(12.33%)	12744(97.88%)	30(0.23%)
postgres	9273	9031(97.39%)	40(0.43%)	9057(97.67%)	14(0.15%)
memcached	233	223(95.7%)	8(3.43%)	229(98.28%)	2(0.85%)
node	20638	13693(66.34%)	530(2.56%)	13486(65.34%)	577(2.79%)
O1					
proftpd	1171	993(84.79%)	14(1.19%)	950(81.12%)	14(1.19%)
vsftpd	391	279(71.35%)	4(1.02%)	256(65.47%)	4(1.02%)
lighttpd	352	296(84.09%)	5(1.42%)	293(83.23%)	6(1.7%)
nginx	1097	758(69.09%)	1(0.09%)	756(68.91%)	1(0.09%)
mysqld	12682	7954(62.71%)	459(3.61%)	7314(57.67%)	442(3.48%)
postgres	9265	7221(77.93%)	717(7.73%)	6932(74.81%)	704(7.59%)
memcached	232	192(82.75%)	15(6.46%)	190(81.89%)	15(6.46%)
node	20638	13693(66.34%)	530(2.56%)	13486(65.34%)	577(2.79%)
O2					
proftpd	1015	853(84.03%)	13(1.28%)	824(81.18%)	14(1.37%)
vsftpd	318	230(72.32%)	3(0.94%)	209(65.72%)	3(0.94%)
lighttpd	290	240(82.75%)	7(2.41%)	239(82.41%)	9(3.1%)
nginx	921	599(65.03%)	0(0.0%)	600(65.14%)	0(0.0%)
mysqld	9742	5714(58.65%)	338(3.46%)	5658(58.07%)	348(3.57%)
postgres	6930	5280(76.19%)	600(8.65%)	5090(73.44%)	593(8.55%)
memcached	133	105(78.94%)	10(7.51%)	104(78.19%)	10(7.51%)
node	20638	13693(66.34%)	530(2.56%)	13486(65.34%)	577(2.79%)
O3					
proftpd	1007	846(84.01%)	13(1.29%)	820(81.42%)	14(1.39%)
vsftpd	314	228(72.61%)	3(0.95%)	207(65.92%)	3(0.95%)
lighttpd	289	240(83.04%)	7(2.42%)	243(84.08%)	7(2.42%)
nginx	908	586(64.53%)	0(0.0%)	588(64.75%)	0(0.0%)
mysqld	9703	5699(58.73%)	324(3.33%)	5645(58.17%)	334(3.44%)
postgres	6848	5221(76.24%)	585(8.54%)	5054(73.8%)	578(8.44%)
memcached	133	105(78.94%)	10(7.51%)	104(78.19%)	10(7.51%)
node	20638	13693(66.34%)	530(2.56%)	13486(65.34%)	577(2.79%)

Table 6.11: The results for calltarget analysis for exp3 and exp4 for the TYPE policy throughout different optimizations

6.1.5 Precision Enhanced Callsite Classification (TYPE)

Experiment Setup To choose the best possible combination operator for the callsite analysis in implementing the TYPE policy, we conducted four experiments based on the data of the two Precision Callsite Classification(COUNT) experiments:

- exp1 intersection combination operator that intersects when both paths are set with a *succ* function that does follow into occurring call functions with backward interprocedural analysis see Table 6.12 for results
- exp2 intersection combination operator that unions when both paths are set with a *succ* function that does follow into occurring call functions with backward interprocedural analysis see Table 6.12 for results
- exp3 union combination operator with a *succ* function that does not follow into occurring call functions with backward interprocedural analysis see Table 6.13 for results
- exp4 general intersection operator with a *succ* function that does not follow into occurring call functions with backward interprocedural analysis see Table 6.13 for results

For each possible version we measured two data points per testtarget, the number and ratio of perfect classifications and the number and ratio of problematic classifications, which in this case refers to overestimations.

We are going to group our results into two classes based on the optimization level, completely unoptimized (O0) and optimized (O1, O2 and O3) as those two groups are highly different in the values they produce.

Results for unoptimized targets

Results for optimized targets

6 Evaluation

O0		exp1		exp2	
Target	#	perfect	problem	perfect	problem
proftpd	174	106(60.91%)	1(0.57%)	106(60.91%)	1(0.57%)
vsftpd	2	1(50.0%)	0(0.0%)	1(50.0%)	0(0.0%)
lighttpd	68	15(22.05%)	0(0.0%)	15(22.05%)	0(0.0%)
nginx	283	130(45.93%)	44(15.54%)	130(45.93%)	44(15.54%)
mysqld	6464	632(9.77%)	383(5.92%)	632(9.77%)	383(5.92%)
postgres	688	236(34.3%)	46(6.68%)	236(34.3%)	46(6.68%)
memcached	47	2(4.25%)	4(8.51%)	2(4.25%)	4(8.51%)
node	8599	4829(56.15%)	1227(14.26%)	4828(56.14%)	1227(14.26%)
O1					
proftpd	70	57(81.42%)	7(10.0%)	57(81.42%)	7(10.0%)
vsftpd	2	2(100.0%)	0(0.0%)	2(100.0%)	0(0.0%)
lighttpd	52	36(69.23%)	10(19.23%)	36(69.23%)	10(19.23%)
nginx	216	144(66.66%)	54(25.0%)	144(66.66%)	54(25.0%)
mysqld	5401	3641(67.41%)	555(10.27%)	3638(67.35%)	555(10.27%)
postgres	443	338(76.29%)	30(6.77%)	337(76.07%)	30(6.77%)
memcached	44	4(9.09%)	7(15.9%)	4(9.09%)	7(15.9%)
node	8599	4829(56.15%)	1227(14.26%)	4828(56.14%)	1227(14.26%)
O2					
proftpd	112	100(89.28%)	4(3.57%)	100(89.28%)	4(3.57%)
vsftpd	14	14(100.0%)	0(0.0%)	14(100.0%)	0(0.0%)
lighttpd	48	33(68.75%)	9(18.75%)	33(68.75%)	9(18.75%)
nginx	234	151(64.52%)	59(25.21%)	151(64.52%)	59(25.21%)
mysqld	6671	4189(62.79%)	918(13.76%)	4187(62.76%)	918(13.76%)
postgres	565	443(78.4%)	31(5.48%)	442(78.23%)	31(5.48%)
memcached	47	5(10.63%)	10(21.27%)	5(10.63%)	10(21.27%)
node	8599	4829(56.15%)	1227(14.26%)	4828(56.14%)	1227(14.26%)
O3					
proftpd	136	122(89.7%)	6(4.41%)	122(89.7%)	6(4.41%)
vsftpd	18	18(100.0%)	0(0.0%)	18(100.0%)	0(0.0%)
lighttpd	49	34(69.38%)	9(18.36%)	34(69.38%)	9(18.36%)
nginx	240	153(63.75%)	62(25.83%)	153(63.75%)	62(25.83%)
mysqld	6946	4358(62.74%)	986(14.19%)	4356(62.71%)	986(14.19%)
postgres	613	492(80.26%)	30(4.89%)	491(80.09%)	30(4.89%)
memcached	47	5(10.63%)	10(21.27%)	5(10.63%)	10(21.27%)
node	8599	4829(56.15%)	1227(14.26%)	4828(56.14%)	1227(14.26%)

Table 6.12: The results for enhanced callsite analysis for exp1 and exp2 for the TYPE policy throughout different optimizations

O0		exp3		exp4	
Target	#	perfect	problem	perfect	problem
proftpd	174	63(36.2%)	1(0.57%)	106(60.91%)	1(0.57%)
vsftpd	2	1(50.0%)	0(0.0%)	1(50.0%)	0(0.0%)
lighttpd	68	1(1.47%)	0(0.0%)	15(22.05%)	0(0.0%)
nginx	283	32(11.3%)	44(15.54%)	130(45.93%)	44(15.54%)
mysqld	6464	344(5.32%)	383(5.92%)	632(9.77%)	383(5.92%)
postgres	688	81(11.77%)	46(6.68%)	236(34.3%)	46(6.68%)
memcached	47	0(0.0%)	4(8.51%)	2(4.25%)	4(8.51%)
node	8599	1664(19.35%)	871(10.12%)	4829(56.15%)	1227(14.26%)
O1					
proftpd	70	25(35.71%)	0(0.0%)	57(81.42%)	7(10.0%)
vsftpd	2	0(0.0%)	0(0.0%)	2(100.0%)	0(0.0%)
lighttpd	52	13(25.0%)	1(1.92%)	36(69.23%)	10(19.23%)
nginx	216	53(24.53%)	34(15.74%)	144(66.66%)	54(25.0%)
mysqld	5401	1196(22.14%)	287(5.31%)	3641(67.41%)	555(10.27%)
postgres	443	127(28.66%)	4(0.9%)	338(76.29%)	30(6.77%)
memcached	44	0(0.0%)	7(15.9%)	4(9.09%)	7(15.9%)
node	8599	1650(19.18%)	868(10.09%)	4829(56.15%)	1227(14.26%)
O2					
proftpd	112	56(50.0%)	0(0.0%)	100(89.28%)	4(3.57%)
vsftpd	14	4(28.57%)	0(0.0%)	14(100.0%)	0(0.0%)
lighttpd	48	14(29.16%)	1(2.08%)	33(68.75%)	9(18.75%)
nginx	234	57(24.35%)	38(16.23%)	151(64.52%)	59(25.21%)
mysqld	6671	1602(24.01%)	418(6.26%)	4189(62.79%)	918(13.76%)
postgres	565	154(27.25%)	1(0.17%)	443(78.4%)	31(5.48%)
memcached	47	0(0.0%)	10(21.27%)	5(10.63%)	10(21.27%)
node	8599	1659(19.29%)	876(10.18%)	4829(56.15%)	1227(14.26%)
O3					
proftpd	136	74(54.41%)	0(0.0%)	122(89.7%)	6(4.41%)
vsftpd	18	6(33.33%)	0(0.0%)	18(100.0%)	0(0.0%)
lighttpd	49	14(28.57%)	1(2.04%)	34(69.38%)	9(18.36%)
nginx	240	65(27.08%)	40(16.66%)	153(63.75%)	62(25.83%)
mysqld	6946	1696(24.41%)	433(6.23%)	4358(62.74%)	986(14.19%)
postgres	613	179(29.2%)	1(0.16%)	492(80.26%)	30(4.89%)
memcached	47	0(0.0%)	10(21.27%)	5(10.63%)	10(21.27%)
node	8599	1643(19.1%)	869(10.1%)	4829(56.15%)	1227(14.26%)

Table 6.13: The results for enhanced callsite analysis for exp3 and exp4 for the TYPE policy throughout different optimizations

6.2 Effectiveness of TypeShield

6.2.1 Theoretical Limit for the Reduction of Calltargets per Callsite

O0 Target	AT	COUNT*			TYPE*		
		limit (avg \pm σ)			median	limit (avg \pm σ)	
,command=node	8599	1	\pm	0.01	20638	4193	\pm 20.31
							O1
proftpd	1171	926.04	\pm	221.56	948.0	883.8	\pm 240.2
vsftpd	391	302.0	\pm	72.0	302.0	161.0	\pm 69.0
lighttpd	352	221.73	\pm	80.81	155.0	205.63	\pm 73.47
nginx	1097	624.31	\pm	294.24	638.0	624.31	\pm 294.24
mysqld	12682	7951.85	\pm	2601.33	5583.0	7436.54	\pm 2305.66
postgres	9265	6546.99	\pm	1688.43	6902.0	5896.25	\pm 2003.95
memcached	232	213.15	\pm	22.24	228.0	154.84	\pm 17.5
node	20638	12841.45	\pm	4441.18	14062.0	10977.88	\pm 4143.54
O2							
proftpd	1015	835.01	\pm	167.84	920.0	783.55	\pm 188.11
vsftpd	318	212.85	\pm	58.27	176.0	148.28	\pm 43.82
lighttpd	290	182.93	\pm	70.83	223.0	167.39	\pm 65.47
nginx	921	520.19	\pm	243.08	518.0	520.19	\pm 243.08
mysqld	9742	6065.19	\pm	1985.69	4304.0	5658.68	\pm 1749.55
postgres	6930	4654.53	\pm	1641.72	5161.0	4221.78	\pm 1776.03
memcached	133	117.14	\pm	16.02	129.0	79.93	\pm 11.99
node	20638	12841.45	\pm	4441.18	14062.0	10977.88	\pm 4143.54
O3							
proftpd	1007	831.04	\pm	160.34	912.0	777.46	\pm 200.02
vsftpd	314	217.33	\pm	59.86	175.0	143.0	\pm 45.25
lighttpd	289	183.22	\pm	69.86	222.0	166.46	\pm 64.46
nginx	908	514.2	\pm	240.9	508.0	514.2	\pm 240.9
mysqld	9703	6064.58	\pm	1986.4	4294.0	5654.4	\pm 1756.96
postgres	6848	4652.45	\pm	1579.86	5108.0	4245.49	\pm 1724.2
memcached	133	117.14	\pm	16.02	129.0	79.93	\pm 11.99
node	20638	12841.45	\pm	4441.18	14062.0	10977.88	\pm 4143.54

Table 6.14: The results of comparing theoretical limits for the different restriction policies throughout different optimizations

6.2.2 TypeShield vs Theoretical Limit

COUNT policy

O0	AT	COUNT safe			COUNT prec		
Target		limit (avg \pm σ)		median	limit (avg \pm σ)		median
,command=node	20638	18232.43	\pm 2452.4	14996.0	15325.62	\pm 4407.39	14996.0
proftpd	1171	1107.32	\pm 31.15	1081.0	923.58	\pm 263.87	964.0
vsftpd	391	379.0	\pm 4.0	379.0	313.0	\pm 62.0	313.0
lighttpd	352	330.59	\pm 10.28	325.0	229.63	\pm 109.36	352.0
nginx	1097	960.72	\pm 241.52	1083.0	654.71	\pm 315.87	591.0
mysqld	12682	11276.47	\pm 1292.4	11091.0	9194.31	\pm 2685.58	9299.0
postgres	9265	8140.28	\pm 708.58	8047.0	6699.76	\pm 2007.44	7026.0
memcached	232	224.56	\pm 6.27	228.0	219.38	\pm 15.4	228.0
node	20638	18233.35	\pm 2447.49	14996.0	15325.62	\pm 4407.39	14996.0
O2							
proftpd	1015	954.41	\pm 27.08	961.0	862.14	\pm 158.04	935.0
vsftpd	318	298.14	\pm 29.45	306.0	224.57	\pm 51.49	192.0
lighttpd	290	270.64	\pm 23.16	286.0	192.18	\pm 96.06	227.0
nginx	921	801.91	\pm 210.01	913.0	559.46	\pm 262.55	913.0
mysqld	9742	8560.9	\pm 1323.59	8550.0	6889.73	\pm 2289.89	9742.0
postgres	6930	6074.52	\pm 525.69	6012.0	4800.62	\pm 1718.52	5262.0
memcached	133	126.08	\pm 5.04	129.0	120.91	\pm 12.58	129.0
node	20638	18221.13	\pm 2479.72	14996.0	15325.62	\pm 4407.39	14996.0
O3							
proftpd	1007	940.87	\pm 37.63	927.0	850.33	\pm 167.8	927.0
vsftpd	314	296.83	\pm 26.16	303.0	227.0	\pm 53.74	189.0
lighttpd	289	268.93	\pm 22.94	284.0	192.0	\pm 94.58	226.0
nginx	908	779.63	\pm 215.74	900.0	555.54	\pm 259.3	900.0
mysqld	9703	8528.42	\pm 1308.98	8516.0	6843.91	\pm 2298.87	9703.0
postgres	6848	5985.35	\pm 509.83	5942.0	4789.06	\pm 1645.87	5211.0
memcached	133	126.08	\pm 5.04	129.0	120.91	\pm 12.58	129.0
node	20638	18232.43	\pm 2452.4	14996.0	15325.62	\pm 4407.39	14996.0

Table 6.15: The results of comparing COUNT safe and precision implementation throughout different optimizations

TYPE policy

O0 Target	AT	TYPE safe			TYPE prec		
		limit (avg \pm σ)		median	limit (avg \pm σ)		median
,command=node	20638	16585.32	\pm 3540.84	17944.0	13928.35	\pm 4307.4	14996.0 O1
proftpd	1171	1055.98	\pm 143.19	1138.0	885.92	\pm 272.94	964.0
vsftpd	391	266.0	\pm 109.0	266.0	203.5	\pm 47.5	203.0
lighttpd	352	308.11	\pm 55.24	352.0	211.53	\pm 106.58	352.0
nginx	1097	881.37	\pm 285.65	913.0	583.75	\pm 283.35	743.0
mysqld	12682	10727.18	\pm 1638.6	12682.0	8730.94	\pm 2517.67	2924.0
postgres	9265	7508.15	\pm 1642.85	8047.0	6112.75	\pm 2158.75	7026.0
memcached	232	179.11	\pm 22.83	168.0	175.04	\pm 19.75	168.0
node	20638	16578.76	\pm 3544.12	17944.0	13928.35	\pm 4307.4	14996.0
O2							
proftpd	1015	895.17	\pm 122.6	935.0	816.77	\pm 176.59	935.0
vsftpd	318	246.14	\pm 81.88	306.0	172.85	\pm 30.26	192.0
lighttpd	290	249.08	\pm 53.1	152.0	174.18	\pm 93.65	148.0
nginx	921	732.73	\pm 238.77	618.0	503.72	\pm 234.88	618.0
mysqld	9742	8082.58	\pm 1528.73	9742.0	6505.05	\pm 2143.58	2155.0
postgres	6930	5678.06	\pm 1157.81	6012.0	4443.7	\pm 1781.19	5262.0
memcached	133	96.23	\pm 14.3	90.0	92.55	\pm 12.8	90.0
node	20638	16558.62	\pm 3566.3	17944.0	13928.35	\pm 4307.4	14996.0
O3							
proftpd	1007	877.16	\pm 145.32	927.0	802.66	\pm 199.76	927.0
vsftpd	314	236.66	\pm 84.39	303.0	167.0	\pm 31.11	189.0
lighttpd	289	245.75	\pm 53.85	152.0	172.93	\pm 92.14	148.0
nginx	908	710.95	\pm 238.94	481.0	496.77	\pm 229.29	350.0
mysqld	9703	8053.65	\pm 1516.42	8709.0	6460.57	\pm 2151.52	2150.0
postgres	6848	5608.85	\pm 1116.93	5942.0	4453.99	\pm 1717.95	5211.0
memcached	133	96.23	\pm 14.3	90.0	92.55	\pm 12.8	90.0
node	20638	16585.32	\pm 3540.84	17944.0	13928.35	\pm 4307.4	14996.0

Table 6.16: The results of comparing TYPE safe and precision implementation throughout different optimizations

6.2.3 Effect of our AddressTaken Analysis

6.3 Security Analysis of TypeShield

In this section, we discuss how effective *TypeShield* is stopping advance code-reuse attacks (CRAs). Patching Policies Two types of diagrams. Table 5 from TypeArmor and a CDF to compare param count and param type. (baseline) here we put the CDF graphs from. There is no accurate security metrics to asses the security level of the enforced policy.

6.4 Mitigation of Advanced Code-Reuse Attacks

We look at our implementation conceptually and assess qualitatively whether our implementation can interfere with various classes of attacks.

6.4.1 COOP

6.4.2 COOP Extensions

6.4.3 Pure Data-only Attacks

O0	AT	COUNT*			TYPE*		
Target		limit (avg \pm σ)		median	limit (avg \pm σ)		median
proftpd	394	346.16	\pm 57.93	353.0	331.64	\pm 65.33	353.0
vsftpd	10	8.0	\pm 2.0	8.0	5.0	\pm 1.0	5.0
lighttpd	59	34.07	\pm 14.73	21.0	31.61	\pm 13.15	21.0
nginx	544	314.71	\pm 149.58	267.0	314.71	\pm 149.58	267.0
mysqld	5940	4232.81	\pm 1083.11	4412.0	3975.87	\pm 989.95	4412.0
postgres	2521	2230.56	\pm 409.18	2311.0	2100.66	\pm 594.48	2311.0
memcached	14	12.27	\pm 2.35	14.0	10.27	\pm 0.96	11.0
node	7524	5065.06	\pm 1547.56	5518.0	4359.79	\pm 1498.04	5518.0
O1							
proftpd	391	340.31	\pm 73.09	352.0	326.5	\pm 88.13	352.0
vsftpd	10	8.0	\pm 2.0	8.0	5.0	\pm 1.0	5.0
lighttpd	59	35.11	\pm 15.19	21.0	32.98	\pm 13.93	21.0
nginx	544	318.94	\pm 151.36	267.0	318.94	\pm 151.36	267.0
mysqld	5917	4149.06	\pm 1057.63	3195.0	3934.67	\pm 945.29	3195.0
postgres	2506	2234.23	\pm 340.85	2298.0	2056.97	\pm 620.02	2298.0
memcached	15	13.5	\pm 2.27	15.0	11.36	\pm 0.93	12.0
node	7526	5067.06	\pm 1547.57	5520.0	4361.77	\pm 1498.09	5520.0
O2							
proftpd	390	348.55	\pm 59.5	369.0	333.11	\pm 72.18	369.0
vsftpd	10	7.14	\pm 1.8	6.0	5.42	\pm 0.9	6.0
lighttpd	59	36.87	\pm 15.13	47.0	34.55	\pm 14.03	33.0
nginx	543	313.87	\pm 150.2	266.0	313.87	\pm 150.2	266.0
mysqld	5879	4103.38	\pm 1054.55	3167.0	3882.72	\pm 937.79	3167.0
postgres	2490	2056.06	\pm 670.46	2284.0	1885.18	\pm 812.27	2284.0
memcached	14	12.27	\pm 2.35	14.0	10.27	\pm 0.96	11.0
node	7528	5068.77	\pm 1547.25	5522.0	4363.56	\pm 1497.97	5522.0
O3							
proftpd	391	349.95	\pm 55.13	369.0	330.55	\pm 80.68	369.0
vsftpd	10	7.33	\pm 1.88	6.0	5.33	\pm 0.94	6.0
lighttpd	59	37.08	\pm 15.04	47.0	34.52	\pm 13.89	33.0
nginx	544	315.23	\pm 151.02	266.0	315.23	\pm 151.02	266.0
mysqld	5873	4112.56	\pm 1057.19	3169.0	3889.42	\pm 946.51	3169.0
postgres	2492	2076.41	\pm 648.38	2285.0	1913.47	\pm 791.78	2285.0
memcached	14	12.27	\pm 2.35	14.0	10.27	\pm 0.96	11.0
node	7524	5065.37	\pm 1547.65	5519.0	4359.99	\pm 1498.21	5519.0

Table 6.17: The results of comparing theoretical limits for the different restriction policies restricted using an address taken analysis throughout different optimizations

O0 Target	AT	COUNT safe			COUNT prec		
		limit (avg \pm σ)		median	limit (avg \pm σ)		median
proftpd	394	290.02	\pm 37.31	283.0	272.36	\pm 55.35	264.0
vsftpd	10	10.0	\pm 0.0	10.0	10.0	\pm 0.0	10.0
lighttpd	59	58.11	\pm 2.12	53.0	53.67	\pm 11.67	21.0
nginx	544	483.15	\pm 96.3	511.0	371.84	\pm 155.48	511.0
mysqld	5940	5345.94	\pm 338.42	5177.0	5225.44	\pm 470.96	5177.0
postgres	2521	2454.68	\pm 201.15	2498.0	2360.18	\pm 350.61	2498.0
memcached	14	13.87	\pm 0.33	14.0	13.7	\pm 1.02	14.0
node	7524	6946.64	\pm 769.8	5937.0	5992.81	\pm 1419.1	5937.0
O1							
proftpd	391	385.68	\pm 3.12	383.0	342.08	\pm 87.9	363.0
vsftpd	10	10.0	\pm 0.0	10.0	9.0	\pm 1.0	9.0
lighttpd	59	55.15	\pm 2.1	54.0	37.65	\pm 20.53	59.0
nginx	544	495.58	\pm 115.68	544.0	347.6	\pm 152.56	305.0
mysqld	5917	5553.05	\pm 480.22	5549.0	4714.82	\pm 1094.78	4858.0
postgres	2506	2447.85	\pm 59.68	2447.0	2220.39	\pm 479.96	2348.0
memcached	15	14.72	\pm 0.44	15.0	14.31	\pm 1.2	15.0
node	7526	6952.22	\pm 763.61	5938.0	5994.27	\pm 1419.58	5938.0
O2							
proftpd	390	384.17	\pm 3.0	385.0	366.16	\pm 48.39	382.0
vsftpd	10	9.71	\pm 1.03	10.0	7.14	\pm 1.8	6.0
lighttpd	59	55.18	\pm 4.87	59.0	39.31	\pm 21.2	50.0
nginx	543	487.48	\pm 120.95	543.0	347.76	\pm 152.94	543.0
mysqld	5879	5453.45	\pm 640.11	5517.0	4615.36	\pm 1165.18	5879.0
postgres	2490	2430.1	\pm 64.27	2431.0	2083.21	\pm 640.18	2332.0
memcached	14	13.68	\pm 0.46	14.0	13.12	\pm 1.33	14.0
node	7528	6949.22	\pm 774.38	5940.0	5995.79	\pm 1419.43	5940.0
O3							
proftpd	391	383.53	\pm 4.61	382.0	363.66	\pm 55.62	382.0
vsftpd	10	9.77	\pm 0.91	10.0	7.33	\pm 1.88	6.0
lighttpd	59	55.16	\pm 4.82	59.0	39.53	\pm 21.03	50.0
nginx	544	481.42	\pm 126.26	544.0	350.04	\pm 153.62	544.0
mysqld	5873	5449.63	\pm 633.8	5511.0	4600.85	\pm 1176.38	5873.0
postgres	2492	2430.49	\pm 64.24	2433.0	2103.6	\pm 620.77	2333.0
memcached	14	13.68	\pm 0.46	14.0	13.12	\pm 1.33	14.0
node	7524	6950.0	\pm 765.22	5937.0	5992.53	\pm 1419.51	5937.0

Table 6.18: The results of comparing COUNT safe and precision implementation restricted using an address taken analysis throughout different optimizations

O0	AT	TYPE safe			TYPE prec		
Target		limit (avg \pm σ)		median	limit (avg \pm σ)		median
proftpd	394	270.5	\pm 43.22	394.0	254.6	\pm 55.33	228.0
vsftpd	10	7.0	\pm 3.0	7.0	7.0	\pm 3.0	7.0
lighttpd	59	54.26	\pm 7.8	59.0	50.97	\pm 12.89	59.0
nginx	544	411.98	\pm 145.67	256.0	317.77	\pm 150.03	511.0
mysqld	5940	4979.26	\pm 724.06	5177.0	4886.99	\pm 735.71	5177.0
postgres	2521	2306.87	\pm 530.49	2498.0	2218.51	\pm 575.83	2498.0
memcached	14	10.63	\pm 1.08	11.0	10.46	\pm 1.0	11.0
node	7524	6368.91	\pm 1185.15	6954.0	5497.2	\pm 1397.47	5937.0
O1							
proftpd	391	373.47	\pm 46.33	389.0	331.12	\pm 96.19	363.0
vsftpd	10	8.5	\pm 1.5	8.0	7.5	\pm 0.5	7.0
lighttpd	59	52.26	\pm 8.22	59.0	35.07	\pm 20.06	59.0
nginx	544	459.81	\pm 133.12	453.0	315.11	\pm 134.33	363.0
mysqld	5917	5350.34	\pm 623.0	5917.0	4539.43	\pm 1037.69	1581.0
postgres	2506	2305.55	\pm 452.87	2447.0	2081.33	\pm 609.8	2348.0
memcached	15	12.4	\pm 1.15	12.0	11.93	\pm 1.35	12.0
node	7526	6373.63	\pm 1182.92	6956.0	5498.44	\pm 1397.89	5938.0
O2							
proftpd	390	372.68	\pm 37.23	382.0	356.23	\pm 59.79	382.0
vsftpd	10	8.0	\pm 2.72	10.0	5.42	\pm 0.9	6.0
lighttpd	59	51.97	\pm 9.43	38.0	36.39	\pm 20.72	36.0
nginx	543	448.69	\pm 135.53	362.0	316.36	\pm 135.22	362.0
mysqld	5879	5225.39	\pm 747.58	5879.0	4427.54	\pm 1097.93	1885.0
postgres	2490	2297.05	\pm 450.04	2431.0	1954.39	\pm 722.62	2332.0
memcached	14	12.06	\pm 0.8	12.0	11.53	\pm 1.31	12.0
node	7528	6368.22	\pm 1189.75	6957.0	5500.11	\pm 1397.84	5940.0
O3							
proftpd	391	367.8	\pm 52.04	382.0	349.65	\pm 74.68	382.0
vsftpd	10	7.77	\pm 2.81	10.0	5.33	\pm 0.94	6.0
lighttpd	59	51.69	\pm 9.54	38.0	36.38	\pm 20.51	36.0
nginx	544	442.03	\pm 138.26	297.0	315.92	\pm 134.16	213.0
mysqld	5873	5223.54	\pm 742.79	5573.0	4414.0	\pm 1108.9	1883.0
postgres	2492	2302.79	\pm 440.31	2433.0	1980.52	\pm 705.69	2333.0
memcached	14	12.06	\pm 0.8	12.0	11.53	\pm 1.31	12.0
node	7524	6373.78	\pm 1181.81	6954.0	5496.75	\pm 1397.92	5937.0

Table 6.19: The results of comparing TYPE safe and precision implementation restricted using an address taken analysis throughout different optimizations

7 Discussion

We are going to discuss several aspects of interest that came up when comparing against TypeArmor, which are our first two topics of discussion, namely the comparison of results between TypeArmor and TypeShield and whether we found any discrepancies between the data we got from their paper and the data we collected. The second part of our discussion is concerned about the limitations of TypeShield and possible venues of improvement.

7.1 Comparison with TypeArmor

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

7.1.1 Precision of Classification

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

perfect classification of callsites, which although seemingly low, is still 69.74% of our precision regarding parameter counts.

7.1.2 Reduction of Available Calltargets

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

7.2 Discrepancies or Problems

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

7.3 Limitations of TypeShield

First of all, we are limited by the capabilities of the DynInst Instrumentation Environment, the main problem, we are facing here is that non returning functions like `exit` are not detected reliably in some cases, which is why we were not able to test the Pure-FTP server, as it heavily relies on these functions. The problem is that those non returning functions usually appear as a second branch within a function that occurs after the normal control flow, causing basic blocks from the following function to be attributed to the current function. This results in a malformed control flow graph and erroneous attribution of callsites and problematic misclassifications for both calltargets and callsites.

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

7.4 Possible Venues of Improvement

To improve our typeanalysis we see atleast two possibilities, incorporating refined dataflow analysis and expanding the scope to also include memory. The main point of improvement is however not the precision but for now more importantly the reduction of underestimations in the callsite analysis.

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

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

8 Related Work

This Chapter we shortly review the main techniques and tools which are related to *TypeShild*. Section 8.1 gives an overview of tools used to recover type inference from binaries. Section 8.2 depicts several techniques used to mitigate code reuse attacks in general. Finally, Section 8.3 presents several source code, binary and runtime techniques which can be used to mitigate COOP attacks.

8.1 Type-Inference on Executables

Recovering variable types from executable programs is very hard in general for several reasons. First, the quality of the disassembly can vary much from used framework to another. *TypeShild* is based on DynInst and the quality of the executable disassembly fits our needs. For a more comprehensive review on the capabilities of DynInst and other tools we advise the reader to have a look at [16].

Second, alias analysis in binaries is undecidable in theory and intractable in practice [21]. There are several most promising tools such as: Rewards [17], BAP [18], SmartDec [19], and Divine [20]. These tools try with more or less success to recover type information from binary programs with different goals. Typical goals are: *i)* full program reconstruction (binary to code conversion, reversing), *ii)* checking for buffer overflows, *iii)* integer overflows and other types of memory corruptions. For a more exhaustive review of such tools we advise the reader to have a look at the review of Caballero et al. [14]. Interesting to notice is that the code from only a few of these tools is available.

TypeShild is most similar to SmartDec since it uses a similar lattice model to represent the possible variable types.

add here 3-4 sentences here about the differences and commonalities between our tool and smartdec

8.2 Mitigation of Code-Reuse Attacks

In the last couple of years researchers have provided many versions of new Code Reuse Attacks (CRAs). These new attacks were possible since DEP [40]

and ASLR [41] were successfully bypassed mostly based on Return Oriented Programming (ROP) [42, 47, 48] on one hand and on the other hand due to the discovery of new exploitable hardware and software primitives.

ROP started to present itself in the last couple of years in many faceted ways such as: Jump Oriented Programming (JOP) [43, 44, 45] which uses jumps in order to divert the control flow to the next gadget and Call Oriented Programming (COP) [46] which uses calls in order to chain gadgets together. CRAs have many manifestations and it is out of scope of this work to list them all.

On one hand, CRAs can be mitigated in general in the following ways: *(i)* binary instrumentation, *(ii)* source code recompilation and *(iii)* runtime application monitoring. On the other hand, there is a plethora of tools and techniques which try to enforce CFI based primitives in executables, source code and during runtime. Next we briefly present the solution landscape together with the approaches and the techniques on which these are based: *(a)* fine-grained CFI with hardware support, PathArmor [15], *(b)* coarse-grained CFI used for binary instrumentation, CCFIR [37], *(c)* coarse-grained CFI based on binary loader, CFCI [36] *(d)* fine-grained code randomization, O-CFI [35], *(e)* cryptography with hardware support, CCFI [39], *(f)* ROP stack pivoting, PBlocker [34], *(g)* canary based protection, DynaGuard [33], *(h)* runtime and hardware support based on a combination of LBR, PMU and BTS registers CFIGuard [38], and *(i)* source code recompilation with CFI and/or randomization enforcement against JIT-ROP attacks, MCFI [28], RockJIT [29] and PiCFI [30].

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

8.3 Mitigation of Advanced Code-Reuse Attacks

COOP [3], Subversive-C [5] and Recursive-COOP [13] are advanced CRAs since these attacks which can not be addressed: *i)* with shadow stacks techniques (i.e., do not violate the caller/callee convention), *ii)* coarse-grained Control-Flow Integrity (CFI) [26, 27] techniques are useless against these attacks, *iii)* hardware based approaches such as Intel CET [6] can not mitigate this attack for the same reason as in *i)*, and *iv)* with OS-based approaches such as Windows Control Flow Guard [7] since the precomputed CFG does not contain edges for indirect call sites which are explicitly exploited during the COOP attack.

However, the following tools can protect against COOP attacks:

Source code based: indirect callsite targets are checked based on vTable integrity. Different types of CFI policies are used such as in the following tools:

SafeDispatch [2], IFCC/VTM [4] LLVM and GCC compiler. Additionally, the Redactor++ [13] uses randomization vTrust [12] checks call target function signatures, CPI [10] uses a memory safety technique in order to protect against the COOP attack.

There are several source code based tools which can successfully protect against the COOP attack. Such tools are: ShrinkWrap [9], IFCC/VTM [4], SafeDispatch [2], vTrust [12], Redactor++ [13], CPI [10] and the tool presented by Bounov et al. [11]. These tools profit from high precision since they have access to the full semantic context of the program though the scope of the compiler on which they are based. Because of this reason these tools target mostly other types of security problems than binary-based tools address. For example some last advances in compile based protection against code reuse attacks address mainly performance issues. Currently, most of the above presented tools are only forward edge enforcers of fine-grained CFI policies with an overhead from 1% up to 15%.

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

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

Binary based: vTable protection is addressed through binary instrumentation in tools such as: vfGuard [32], vTint [31]. However, none of these tools can help to mitigate against COOP. The only binary based tool which we are aware of that can mitigate protect against COOP is TypeArmor [8]. TypeArmor uses a fine-grained CFI policy based on caller (only indirect call sites)/callee matching which consists in checking during runtime if the number of provided and needed parameters match.

TypeShield is most similar to TypeArmor [8] since we also enforce strong binary-level invariants on the number of function parameters. *TypeShield* similarly to TypeArmor targets exclusive protection against advanced exploitation techniques which can bypass fine-grained CFI schemes and VTable protections at the binary level.

However, *TypeShield* is much more precise than TypeArmor since its enforcing

policy is also based on the types of the function parameters. This results in a more precise selection of caller/callee pairs on which the fine-grained CFI policy is enforced. Thus, we achieve a reduced runtime overhead than TypeArmor since we enforce our CFI policy on fewer caller/callee pairs than TypeArmor.

add here 2-3 sentences here to round up this section, present another interesting fact.

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

IntelCET [6] is based on, `ENDBRANCH`, a new CPU instruction which can be used to enforce an efficient shadow stack mechanism. The shadow stack can be used to check during program execution if caller/return pairs match. Since the COOP attack reuses whole functions as gadgets and does not violate the caller/return convention than the new feature provided by Intel is useless in the face of this attack. Nevertheless other highly notorious CRAs may not be possible after this feature will be implemented main stream in OSs and compilers.

Windows Control Flow Guard [7] is based on a user-space and kernel-space components which by working closely together can enforce an efficient fine-grained CFI policy based on a precomputed CFG. These new feature available in Windows 10 can considerably rise the bar for future attacks but in our opinion advanced CRAs such as COOP are still possible due the typical characteristics of COOP.

PathArmor [15] is yet another tool which is based on a precomputed CFG and on the LBR register which can give a string of 16 up to 32 pairs of from/to addressed of different types of indirect instructions such as `call`, `ret`, and `jump`. Because of the sporadic query of the LBR register (only during invocation of certain function calls) and because of the sheer amount of data which passes through the LBR register this approach has in our opinion a fair potential to catch different types of CRAs but we think that against COOP this tool can not be used. First, because of the fact that the precomputed CFG does not contain edges for all possible indirect call sites which are accessed during runtime and second, the LBR buffer can be easily tricked by adding legitimate indirect call sites during the COOP attack.

9 Future Work

We see at least five different possible ways of future venues of research, which range from improving the structural matching between ground truth and binary based data to expanding our schema altogether by incorporating aliasing and tracking memory operations.

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

Finding a better suited callsite analysis would present itself as another important possibility, as we have still relatively high - up to 16% - number of underestimated callsites. However, this venue should only be attempted after significant improvements to the structural matching of callsites.

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

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

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

10 Conclusion

Advanced code reuse attacks like COOP and its extensions or Control Jujutsu manifest due to a combination of facts and problems, like memory corruption or predictable binary layout and the fact that the larger our binaries get the higher the chance they contain useful gadgets for an attacker. However, due to their nature traditional CFI cannot detect them, as they do not actually replace code to modify the control flow, but change pointers in memory, which redirects the targets of indirect callsites, which are uncertain at the time of compilation. Two of the most common targets are the pointers to virtual function tables to implemented inheritance in C++ and global function pointers. The control flow exhibited by the binary while functioning normal and while under attack will seem the same. Address taken analysis already helped cutting down the number of possible calltargets one could inject by a considerable amount. And typearmor improved on that by implementing invariants for both callsites and calltargets based on the number of parameters. We had no access to their sourcecode and therefore had to rely on their paper to implement an approximation for which we generated comparable results regarding precision. We improved on that solution by implementing Typeshield, which allows for a more fine-grained classification of calltargets and indirect callsites by implementing a rather simplistic register wideness based type analysis. However, as simplistic as that analysis might be, we showed that except for special cases (nginx), we were able to improve upon a parameter count based implementation by reducing the average target count by about 10% to 20%.

Acronyms

CRA	Code Reuse Attack
BLA	kkk

Contents

List of Figures

2.1	Code example used to illustrate how a COOP loop gadget works	6
2.2	Class inheritance hierarchy of the classes involved in the COOP attack against the Firefox browser. Red letters indicate forbidden vTble entries and green letters indicate allowed vTable entries for the given indirect call site contained in the main loop gadget.	10
4.1	<i>count</i> policy classification schema	15
4.2	<i>type</i> policy schema for callsites and calltargets	17
4.3	ASM code of the <code>make_cmd</code> function with optimize level O2, which has a variadic parameter list	24

List of Figures

List of Tables

4.1	Different mappings for combining two liveness state values in horizontal matching for the <i>count</i> policy	22
4.2	The union mapping operator for liveness in the <i>type</i> policy	23
4.3	Different mappings for combining two reaching state values in horizontal matching for the <i>count</i> policy	28
4.4	Different mappings for combining two reaching state values in horizontal matching for the <i>type</i> policy	30
6.1	Table shows the quality of structural matching provided by our automated verify and test environment, regarding callsites and calltargets when compiling with optimization levels O0 through O3. The label clang miss denotes elements not found in the dataset of the clang/llvm pass. The label tool miss denotes elements not found in the dataset of TypeShield	38
6.2	The results for calltarget analysis using the destructive/intersection combination operator for the COUNT policy throughout different optimizations	40
6.3	The results for calltarget analysis using the union combination operator for the COUNT policy throughout different optimizations	41
6.4	The results for callsite analysis using the destructive/intersection combination operator for the COUNT without interprocedural analysis policy throughout different optimizations	43
6.5	The results for callsite analysis using the union combination operator for the COUNT without interprocedural analysis policy throughout different optimizations	44
6.6	The results for callsite analysis using the true union combination operator for the COUNT without interprocedural analysis policy throughout different optimizations	45
6.7	The results for callsite analysis using the destructive/intersection combination operator for the COUNT with interprocedural analysis policy throughout different optimizations	47

6.8	The results for callsite analysis using the union combination operator for the COUNT with interprocedural analysis policy throughout different optimizations	48
6.9	The results for callsite analysis using the true union combination operator for the COUNT with interprocedural analysis policy throughout different optimizations	49
6.10	The results for calltarget analysis for exp1 and exp2 for the TYPE policy throughout different optimizations	51
6.11	The results for calltarget analysis for exp3 and exp4 for the TYPE policy throughout different optimizations	52
6.12	The results for enhanced callsite analysis for exp1 and exp2 for the TYPE policy throughout different optimizations	54
6.13	The results for enhanced callsite analysis for exp3 and exp4 for the TYPE policy throughout different optimizations	55
6.14	The results of comparing theoretical limits for the different restriction policies throughout different optimizations	56
6.15	The results of comparing COUNT safe and precision implementation throughout different optimizations	57
6.16	The results of comparing TYPE safe and precision implementation throughout different optimizations	58
6.17	The results of comparing theoretical limits for the different restriction policies restricted using an address taken analysis throughout different optimizations	60
6.18	The results of comparing COUNT safe and precision implementation restricted using an address taken analysis throughout different optimizations	61
6.19	The results of comparing TYPE safe and precision implementation restricted using an address taken analysis throughout different optimizations	62

Bibliography

- [1] Mingwei Zhang and R. Sekar, “Control Flow Integrity for COTS Binaries”, In *Proceedings of the USENIX conference on Security (USENIX SEC)*, ACM, pp. 337-352, 2013.
- [2] Dongseok Jang, Zachary Tatlock, and Sorin Lerner, “SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks”, In *Symposium on Network and Distributed System Security (NDSS)*, 2014.
- [3] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, “Counterfeit Object-oriented Programming”, In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.
- [4] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike, “Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM”, In *Proceedings of the USENIX conference on Security (USENIX SEC)*, ACM, 2014.
- [5] Julian Lettner, Benjamin Kollenda, Andrei Homescu, Per Larsen, Felix Schuster, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Michael Franz, “Subversive-C: Abusing and Protecting Dynamic Message Dispatch”, In *USENIX Annual Technical Conference (USENIX ATC)*, 2016.
- [6] Intel, “Intel CET, <http://blogs.intel.com/evangelists/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks/>, 2016.
- [7] Microsoft, “Windows Control Flow Guard”, [https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx), 2015.
- [8] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano 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.

- [9] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, H. Bos, “ShrinkWrap: VTable Protection Without Loose Ends”, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.
- [10] Volodymyr Kuznetsov, László Szekeres, Mathias Payer†, George Candea, R. Sekar, Dawn Song, “Code-Pointer Integrity”, In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [11] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner, “Protecting C++ Dynamic Dispatch Through VTable Interleaving”, In *Symposium on aravindNetwork and Distributed System Security (NDSS)*, 2016.
- [12] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer and Dawn Song, “VTrust: Regaining Trust on Virtual Calls”, In *Symposium on Network and Distributed System Security (NDSS)*, 2016.
- [13] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, Michael 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.
- [14] Juan Caballero, and Zhiqiang Lin, “Type Inference on Executables”, In *ACM Computing Surveys (CSUR)*, 2016.
- [15] Victor van der Veen, Dennis Andriesse, Enes Göktas, Ben Gras. Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida, “Practical Context-Sensitive CFI”, In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
- [16] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, “An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries”, In *Proceedings of the USENIX Conference on Security (USENIX SEC)*, 2016.
- [17] Zhiqiang Lin Xiangyu Zhang Dongyan Xu, “Automatic Reverse Engineering of Data Structures from Binary Execution”, In *Symposium on Network and Distributed System Security (NDSS)*, 2010.
- [18] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz, “BAP: A Binary Analysis Platform”, In *Proceedings of Computer Aided Verification (CAV)*, 2011.

- [19] Alexander Fokin, Yegor Derevenets, Alexander Chernov, and Katerina Troshina, "SmartDec: Approaching C++ decompilation", In *Working Conference on Reverse Engineering (WCRE)*, 2011.
- [20] G. Balakrishnan and T. Reps, "DIVINE: Discovering variables in executables", In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2007.
- [21] Alan Mycroft, "Lecture Notes", In <https://www.cl.cam.ac.uk/~am21/papers/sas07slides.pdf>, 2007.
- [22] Andrew R. Bernat and Barton P. Miller, "Anywhere, Any-Time Binary Instrumentation", In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, (PASTE)*, 2011.
- [23] Dynamic Instrumentation Tool Platform "DynamoRIO", In <http://dynamorio.org/home.html>.
- [24] I. JTC1/SC22WG21, "ISO/IEC 14882:2013 Programming Language C++ (N3690)", In <https://isocpp.org/files/papers/N3690.pdf>, 2013.
- [25] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee, "Type Casting Verification: Stopping an Emerging Attack Vector", In *Proceedings of the USENIX Conference on Security (USENIX SEC)*, 2015.
- [26] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti, "Control Flow Integrity", In *the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.
- [27] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti, "Control Flow Integrity Principles, Implementations, and Applications", In *ACM Transactions on Information and System Security (TISSEC)*, 2009.
- [28] Ben Niu, and Gang Tan, "Modular Control-Flow VTint: Protecting Virtual Function TabIntegrity", In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.
- [29] Ben Niu, and Gang Tan, "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity", In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.
- [30] Ben Niu, and Gang Tan, "Per-Input Control-Flow Integrity", In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

- [31] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song, "VTint: Protecting Virtual Function Tables Integrity", In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.
- [32] Aravind Prakash, Xunchao Hu, and Heng Yin, "Strict Protection for Virtual Function Calls in COTS C++ Binaries", In *Symposium on Network and Distributed System Security (NDSS)*, 2015.
- [33] Theofilos Petsios, Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis, "DynaGuard: Armoring Canary-based Protections against Brute-force Attacks", In *Annual Computer Security Applications Conference (ACSAC)*, 2015.
- [34] Aravind Prakash, and Heng Yin, "Defeating ROP Through Denial of Stack Pivot", In *Annual Computer Security Applications Conference (ACSAC)*, 2015.
- [35] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz, "Opaque Control-Flow Integrity", In *Symposium on Network and Distributed System Security (NDSS)*, 2015.
- [36] Mingwei Zhang, and R. Sekar, "Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-ROP Attacks", In *Annual Computer Security Applications Conference (ACSAC)*, 2015.
- [37] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou, "Practical Control Flow Integrity & Randomization for Binary Executables", In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.
- [38] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding, "Hardware-Assisted Fine-Grained Code-Reuse Attack Detection", In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.
- [39] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières, "CCFI: Cryptographically Enforced Control Flow Integrity", In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
- [40] Microsoft, Changes to Functionality in Microsoft Windows XP Service Pack 2., <https://technet.microsoft.com/en-us/library/bb457151.aspx>.
- [41] PaX Team. Address Space Layout Randomization, <https://pax.grsecurity.net/docs/aslr.txt>, 2001.

- [42] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC", In *ACM Conference on Computer and Communications Security (CCS)*, 2008.
- [43] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A New Class of Code-Reuse Attack", In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.
- [44] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns", In *ACM Conference on Computer and Communications Security (CCS)*, 2010.
- [45] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Return-Oriented Programming without Returns on ARM", In *Technical report, Technical Report HGI-TR-2010-002, Ruhr-University Bochum*, 2010.
- [46] Nicholas Carlini, and David Wagner, "ROP is still dangerous: Breaking Modern Defenses", In *Proceedings of the USENIX conference on Security (USENIX SEC)*, ACM, 2014.
- [47] T. Kornau, "Return-Oriented Programming for the ARM Architecture", <http://www.zynamics.com/downloads/kornau-tim-diplomarbeit--rop.pdf>, 2009.
- [48] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (On the x86)", In *ACM Conference on Computer and Communications Security (CCS)*, 2007.