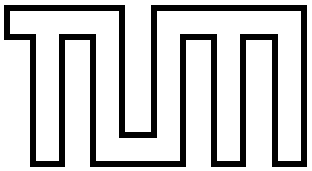


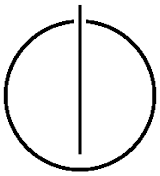
FAKULTÄT FÜR INFORMATIK

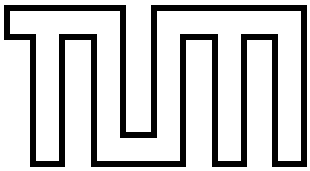
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

to do title eng

Matthias Konstantin Fischer





FAKULTÄT FÜR INFORMATIK

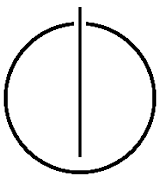
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

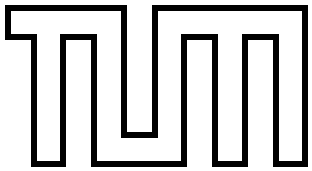
Masterarbeit in Informatik

to do title eng

to do, title ger

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





FAKULTÄT FÜR INFORMATIK

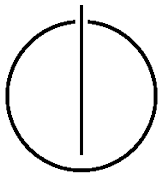
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

to do title eng

to do, title ger

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



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

München, den 24. Oktober 2016 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 and 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 allow 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, vptrs 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 *TypeShild*, 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, *TypeShild* can be automatically and easily used in conjunction with legacy applications or where source code is missing. It achieves higher caller/caller matching (precision) and with reasonable run-time overhead. We have applied *TypeShild* 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 this applications from vptr corruptions vulnerabilities. Our evaluation revealed that *TypeShild* 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

1 Introduction

Control-Flow Integrity (CFI) [34, 31] 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 [9] 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 virtal call sited, 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 (thre is no real application which can really defend against).

TypeArmor [9] is the most similar tool to *TypeShild* and it used function parameter coundting by enforcing a fine-grained policy on valid indirect caller/callee pairs inside a binary. The policy is checked during run-time by cheching that the number of parameters which an indirect call site provides matches with the number of parameter the calle 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/calle 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:

1. We did this

2. We did this
3. We did this.

   The rest of the MA is organized as follows.

1.1 Motivation

here coms the Motivation.
2-3 sentences.

1.2 Research Goals

here coms the Motivation.
2-3 sentences.

1.3 Outline

The remainder of this thesis is organized as follows. We start by giving an overview of how *TypeShild* is designed to mitigate COOP attacks. Chapter 2 gives an overview of bad C++ forward edge calls and its security implications. Chapter 3 contains an high level overview of *TypeShild*. Chapter 4 gives an overview of the techniques used in *TypeShild*. Chapter 5 presents briefly the implementation details of our tool. The *TypeShild* 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 Forbidden C++ Forward Indirect Calls Exposed

2.1 Late Binding in C++

Explain how late binding is implemented in C++ and which role virtual functions play. How is late binding basically implemented.

2.2 Virtual Dispatch in Practice

2.3 Security Implications of Forbidden C++ Forward Indirect Calls

How can Forbidden C++ Forward Indirect Calls exploited? First through COOP attacks, vptr corruption and then fake vtable insertion an reuse or reuse of avaialble v tables.

2.4 Running Example: CVE X

CVE-2014-3176

3 TypeShild Overview

after the Design and Implementation section is done we pick the most important points of TypeShild 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

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 deref- erenced 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

todo

4.1 Static Analysis

todo

4.1.1 Callee Analysis

todo

Forward Analysis

todo

indirect calls todo

returns todo

others todo

Merging Paths

todo

Argument Count

todo

Type Inference

todo

Variadic Functions

todo

Conservativeness

todo

Example of Operation

todo

4.1.2 Callsite Analysis

todo

Backward Analysis

todo

Merging Paths

todo

Argument Count

todo

Type Inference

todo

Compiler Optimizations

todo

Conservativeness

todo

Example of Operation

todo

4.1.3 Return Values

todo

Non-void Callsites

todo

Void Callees

todo

4.2 Runtime Enforcement

todo

4.2.1 Shadow Code Memory Preparation

todo

4.2.2 CFI Enforcement

todo

4.2.3 CFC Enforcement

todo

this the is the old snipets chapter: remove if not needed. ————————

As of now we used the full set of possible calltargets, which is the set of addresses of all function entry basic blocks. To further restrict the possible calltargets per callsite, we explored the notion of incorporating an address taken analysis into our application. The notion is that any indirect control flow instruction might only target addresses that are considered taken. An Address is considered to be a taken address, if it is loaded to memory or a register usually this is a constant, !optional! however it is also possible that simple calculations using multiplication and/or addition are used. We are not concerned with more complex calculations, because we have not observed compilers resorting to more complex methods and literature so far does agree [**?**].

Based on the notions of [**?**], introduced several types of indirect control flow targets of which only !shorthand! Code Pointer Constants (CK) and !shorthand! Computed Code Pointers (CC) are of interest to us. The reason for that is that the others are usually the target of indirect jumps, however we are (as of now) only interested in callsites.

!optional!

Definition 4.1 *!shorthand! Computed Code Pointers (CC) are, addresses that are computed during binary execution. In [**?**] this set only contains targets to intraprocedural indirect jumps and is thus of no interest for us*

Our approach of indentifying taken addresses is a two pronged approach. First, we iterate over the raw binary content of data segments additionally identifying possible dereferencable addresses. Second, we iterate over all instructions in functions within the disassembled binary.

As suggested in [**?**], we slide a !todo!, how much byte ? 4 or 8 ? what happens on X86-64 compared to x86window over the data sections of the binary (namely the .data, the .rodata and the .dynsym).

We rely on Dyninst [**?**] to supply us with the correct function bounadries and addresses of instructions, which we then pass onto our instruction decoder, which is based on DynamoRIO. In essence there are types of analysis that are performed on each instruction. First we identify all relevant constants from the instruction

1. If the instructions is a control flow instructions, we completely ignore it, as it cannot give us any information that is relevant. !todo! can we trace back from memory addresses and registers, what essentially is being called ?

2. We look a the sources and !todo! targets of the instructions and add it to the list of potentially interesting targets

3. If the target is a RIP-based address, we rely on DynamoRIO to decode it and also add it to the list of potentially interesting target
4. !todo what is with constant functions ?
5. !todo! can we have DynamoRIO infer the result of simple lea instructions ?

Then for the resulting set of addresses, we check whether each either point to the entry block of a function, points within the .plt section, or is present in our reference map, which we calculated earlier. !todo! is the reference map needed ?

———————-

5 Implementation

We implemented *TypeShild* based on the DynInst [24] binary Instrumentation framework (revision X, version X). The static instrumentation module is implemented based on DynInst and an additional patch which we added in order to make DynInst to better deal with patching indirect caller and callee pairs. The type inference module is implemented based on DynamoRIO [25]. In total, *TypeShild* is implemented in X lines of C++ code (excluding empty lines and comments). *TypeShild* is at this stage of development implemented for the Linux x86-64 bit platform, but it is important to notice that there are no platform-dependent APIs used. For example, *TypeShild* uses for instrumentation DynInst and for the type inference of the function parameter variables DynamoRIO. As all theses platform-dependent features can be used on other platforms, we believe that *TypeShild* can be easily ported to other platforms as well.

TypeShild uses an effective mechanism for path merging which allows our tool to ... please complete the sentence.

TypeShild "mention another main characteristic about our tool similar to the one above"

TypeShild "mention yet another mai characteristic about our tool similar to the one above, if there is one."

example: C A V ER also maintains the top and bottom addresses of stack segments to efficiently check pointer membership on the stack.

We also changed the DynInst patching mechanism in oder to facilitate blaa. " please finish sentence".

@Matthias: add numeric values at the end. and the missing point from above. The best would be if the Implementation chapter would be exact a page.! Please accomplish this.

6 Evaluation

We evaluated our tool X with Y popular servers, by instrumenting them with our tool. We performed runtime performance test with the following applications.

Our Evaluation aims to answer the following research questions:

- **R1:** How effective is *TypeShild* in securing binary programs against the COOP attack?
- **R2:** How precise is *TypeShild* in detecting the types of the caller/caller pairs?
- **R3:** What is the instrumentation overhead of *TypeShild*?
- **R4:** What is the performance overhead of *TypeShild*?
- **R5:** What is the protection coverage of *TypeShild*? How many caller/called pairs are secured by *TypeShild* and how many remain unsecured?
- **R6:** What level of security does *TypeShild* offer for a protected executable? see typeArmor paper VIII Sec. Analysis.
- **R7:** Against which kind of attacks can *TypeShild* secure executables?

Comparison methods. Example: We used UBSAN (compare with TypeArmor), the state-of-art tool for detecting bad-casting bugs, as our comparison tar- get of C A V ER . Also, We used C A V ER - NAIVE , which dis- abled the two optimization techniques described in §4.4, to show their effectiveness on runtime performance opti- mization.

Experimental setup. Example: All experiments were run on Ubuntu 13.10 (Linux Kernel 3.11) with a quad-core 3.40 GHz CPU (Intel Xeon E3-1245), 16 GB RAM, and 1 TB SSD-based storage.

6.1 R1: Effectiveness of our Tool

6.1.1 Mitigation of Advanced Code-Reuse Attacks

In this section, we discuss how effective *TypeShild* is stopping advance code-reuse attacks (CRAs).

Table 6.1: Classification CS

Exploit	Stopped?	Notes
X	X	X

6.1.2 Effectiveness against COOP

6.1.3 Stopping COOP Exploits in Practice

6.1.4 Control Jujutsu

yes no?

6.1.5 COOP Extensions

6.1.6 Pure Data-only Attacks

Table 6.2: Classification CS

target	opt	#CS	problems	0	-1	-2	-3	-4	-5	-6	non-void-ok	non-void-probl.
x	x	x	x	x	x	x	x	x	x	x	x	x

Table 6.3: Compound

opt	#CS	cs args (perfect %)	cs args (problem %)	cs non-void (correct %)	cs non-void (probl. %)	#ct	ct args (perfect %)	ct args (probl. %)	ct void (correct %)	ct void (correct %)
x	x	x	x	x	x	x	x	x	x	x

Table 6.4: Classification CT

target	opt	#CS	problems	0	-1	-2	-3	-4	-5	-6	non-void-ok	non-void-probl.
x	x	x	x	x	x	x	x	x	x	x	x	x

Table 6.5: Callsite Classification for paramcount

target	opt	problematic	+0	+1	+2	+3	+4	+5	+6
x	x	x	x	x	x	x	x	x	x

Table 6.6: Calltarget Classification

target	opt	problematic	-0	-1	-2	-3	-4	-5	-6
x	x	x	x	x	x	x	x	x	x

Table 6.7: Coumpound table

target	opt	#	Callsites: param perf. %, probl %	#	Callsites: param perf. %, probl %
x	x	x	x	x	x

Table 6.8: MAtching table

target	opt	ct	Ct probl.	at	at prob.	cs	clang cs probl.	padyn cs probl.
x	x	x	x	x	x	x	x	x

Table 6.9: policy evaluation

target	opt	policy	0	1	2	3	4	5	6	sumarry
x	x	x	x	x	x	x	x	x	x	x

Table 6.10: param wideness

5	4	3	2	1	0	param/wideness
x	x	x	x	x	x	0
x	x	x	x	x	x	8
x	x	x	x	x	x	16
x	x	x	x	x	x	32
x	x	x	x	x	x	64

Table 6.11: tabelle 7

5	4	3	2	1	0	param/wideness
x	x	x	x	x	x	0
x	x	x	x	x	x	8
x	x	x	x	x	x	16
x	x	x	x	x	x	32
x	x	x	x	x	x	64

Table 6.12: tabelle 7

5	4	3	2	1	0	param/wideness
x	x	x	x	x	x	0
x	x	x	x	x	x	8
x	x	x	x	x	x	16
x	x	x	x	x	x	32
x	x	x	x	x	x	64

alternative for abobe:

Table 6.13: matching

target	opt	fn_count	fn_problem	at_count	at_problem	cs_count	cs_clang	cs_padyn
x	x	x	x	x	x	0	0	0

Figure 6.1: impact of CFI and CFC

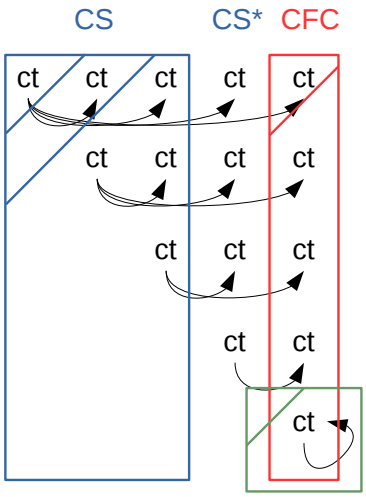


Figure 6.2: liveness iteration, dummy

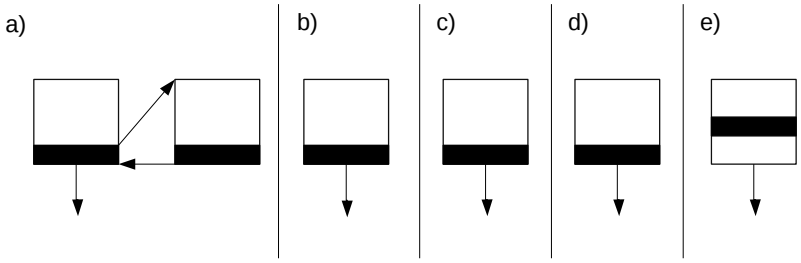


Table 6.14: pairings compares

target	opt	policy	0	1	2	3	4	5	6	summary
proftpd	x	x	x	x	x	0	0	0	0	0
proftpd	x	x	x	x	x	0	0	0	0	0
vsftpd	x	x	x	x	x	0	0	0	0	0
vsftpd	x	x	x	x	x	0	0	0	0	0

Figure 6.3: reaching iteration, dummy

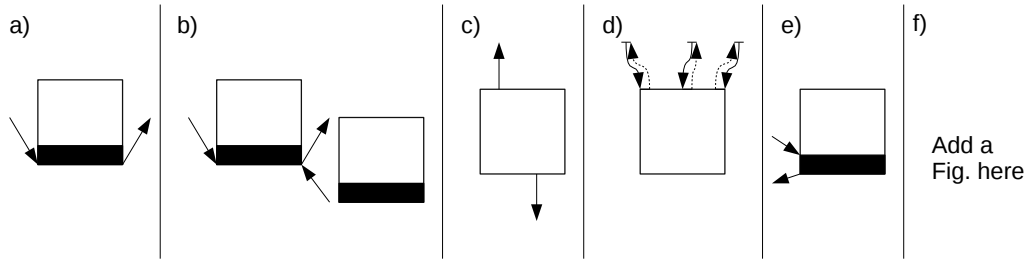


Table 6.15: policy baseline

target	opt	policy	0	1	2	3	4	5	6	summary
proftpd	x	x	x	x	x	0	0	0	0	0
proftpd	x	x	x	x	x	0	0	0	0	0
vsftpd	x	x	x	x	x	0	0	0	0	0
vsftpd	x	x	x	x	x	0	0	0	0	0

6.2 R2: Precision of our Tool

here we have to show how precise is our tool and compare it to TypeArmor. Precision means. Correct identified Callsites and Callees with respect to number of params and types of the params.

Classification Callsites

this has to do with precision of our tool! So put in the right R from above.

overestimation param count. table. number of parameters.

Calltargets underestimation param table.

AT ParamCount table, cdf, baseline vs. server. approximations.

ParamType has to do with precision and effectiveness so move it up in the right R from above!

table, cdf, baseline vs. server. approximations.

this has to do with precision of our tool! So put in the right R from above.

6.3 R3: Instrumentation overhead of our Tool

Here we measure how much the binaries increased in size after the instrumentation was added to the binaries.

6.4 R4: Performance overhead of our Tool

Here we measure how much performance overhead the instrumeentation incurs. Here we measure with the same SPEC2006 programs that was used in the TypeArmor paper. spec 2006.

Table 6.16: Performance Tabble, more better is a bar chart here!

target	opt	problematic	+0	+1	+2	+3	+4	+5	+6
x	x	x	x	x	x	x	x	x	x

6.5 R5: Protection Coverage

here we will compare *TypeShild* with TypeArmor w.r.t. the protection coverage during instrumentation.

Table 6.17: TypeShild vs TypeArmor

Spec Prog. Name	number of checks added at the callsite and calle TypeShild vs TypeArmor	n
x	x	x

6.6 R6: Security Analysis

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.

6.7 R7: Which kind of attacks can our tool defend off

Basically it protects against COOP and other vptr corruption attacks. Also does it protect against Control Jujutsu?

7 Discussion

We have to define which points make sense and then talk about each other
Suggestion:

7.1 How to make the type inference more precise?

1/2 page

7.2 Comparison with TypeArmor and why are we better than TypeArmor?

1/2 page

7.3 Whys is not TypeArmor working as it should to?

1/2 page

7.4 What is not clear bout TypeArmor?

1/2 page

7.5 What can for sure not work as in TypeArmor paper explained?

Furthermore, bla ...

7.6 What are the limitations of TypeShild?

what can out tool achieve and what can not be achieved by our tool? I suggest adding all the problem entries from the above tables here and briefly discuss them here.

8 Related Work

This chapter we shortly review the main techniques and tools which are related to *TypeShild*. Section 8.1 presents several techniques which can not fully defend against advanced CRAs. Section 8.2 gives an overview of tools used to recover type inference from binaries. Section 8.3 highlights several tools which can mitigate against the COOP attack on the binary level. Section 8.4 presents tools which can mitigate against the COOP source code level. Finaly, section 8.5 shows some of the most promissing runtime-based mitigation techniques against COOP.

8.1 Mitigation of Advanced Code-Reuse Attacks

COOP [3], Subversive-C [6] and Recursive-COOP [5] are advanced CRAs since these attacks: *i)* can not be addressed with shadow stacks techniques (i.e., do not violate the caller/calle convention), *ii)* coarse-grained CFI techniques are useless against these attacks, *iii)* hardware based approaches such as Intel CET [7] can not mitigate this attack for the same reason as in *i)*, and *iv)* OS-based approaches such as Windows Control Flow Guard [8] does not defend against COOP since the precomputed CFG does not contain edges for indirect call sites which are explicitly exploited during the COOP attack.

CRAs have many manifestations and it is out of scope of this work to list them all here. CRAs can be addressed in general the following ways: *(i)* binary instrumentation, *(ii)* source code recompilation and *(iii)* runtime application monitoring. While there is a pletora of tools and techniques which try to enforce CFI primitives in executables, source code and during runtime, we briefly list few of them in order for the reader to get familiar with the solution landscape. The approaches used to combat against CRAs are roughly based on the following techniques: *(a)* fine-grained CFI with hardware support, PathArmor [17], ((b)) coarse-grained CFI used for binary instrumentation, CCFIR [40], *(c)* coarse-grained CFI based on binary loader, CFCI [41] *(d)* fine-grained code randomization, O-CFI [38], *(e)* cryptografy with hardware support, CCFI [37], *(f)* ROP stack pivoting, PBlocker [43], *(g)* canary based protection, DynaGuard [42], *(h)* checking vTable integrity for protecting against COOP based on CFI for source code auch as SafeDispatch [2], vtv [4] LLVM and GCC compiler based vor vTable protection and binary rewriting such as vfGuard [44], vTint [39] and [5], *(i)* runtime and hardware support based on a combination of LBR, PMU and BTS registers CFIGuard [29] and *(j)* source code recompilation with CFI and/or randomization enforcement against JIT-ROP attacks, MCFI [28], RockJIT [32] and PiCFI [33].

Notice that the above techniques are useless against the aforementioned advanced CRAs, the list is not exaustive and new protection technies arise by combining available techniques or by using newly available hardware features.

8.2 Tyepe-Inference on Executables

Recovering variable types from executable programs is very hard in general for several reasons. First, the quality of the disasembly can very much from used framework to another. *TypeShild* is based on DynInst and the quality of the execuatble disasembly fits our needs. For a more comprehensive review on the capabilities of DynInst and other tools we advice the reader to have a look at [18].

Second, alias analysis in binares is undecidable in theory and intractable in practice [23]. There are several most promising tools such as: Rewards [19], BAP [20], SmartDec [21], and Divine [22]. 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 convertion, reversing), *ii)* checking for buffer overflows, *iii)* integer overflows and other types of memory corruptions. For a more exhaustive review of such tools we advice the reader to have a look at the review of Caballero et al. [16]. Intresting to notice is that the code from only a few of this 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

8.3 Binary-based Protection against COOP

TypeShild is most similar to TypeArmor [9] since we also enforce strong binary-level invariants on the number of function paramters. *TypeShild* 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, *TypeShild* 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/calle pairs on which the fine-grained CFI policy is enforced. Thus, we achieve a reduced runtime overhead than TypeArmor since we enfoce 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.

8.4 Source Code based Protection against COOP

There are several source code based tools which can succesfully protect against the COOP attack. Such tools are: ShrinkWrap [10], IFCC/VTV [4], SafeDispatch [2], vTrust [13], Readactor++ [15], CPI [11] and the tool presented by Bounov et al. [12]. These tools profit from high precision since they have access to the full semantic context of the program though the scope of the compiler on which they are based. Because of this reason these tools target mostly other types of security problems than binary-based tools address. For example some last advancec 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 possbile semantic information from an executable in order to make the program run as fast as possible. In light of this fact, *TypeShild* 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 *TypeShild* are suffiecient to mitigate all versions of the COOP attack we take a more conservative path by claiming that *TypeShild* further raises the bar w.r.t. what is possible when defending against COOP attacks on the binary level.

8.5 Runtime-based Protection against COOP

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

IntelCET [7] 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 pais match. Since the COOP attack reuses whole functions as gadgets and does not violate the caller/return convention than the new feature provided by interl 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 [8] is based on a user-space and kernel-space components which by working closelly 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 [17] 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 thorugh the LBR register this apporach 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 triked by adding legitimate indirect call sites during the COOP attack.

9 Future Work

This chapter presents in section 9.1 the future steps in order to improve the precision and efficiency of *TypeShild*.

9.1 Future Work

In future we want to address the following points in order to increase the precision, efficiency and coverage of *TypeShild*.

Type inference. The type inference precision of function parameters in *TypeShild* can be increased by blaaa....

to do

Function parameter counting. The counting of function parameters can be made more reliable by tackling the following points.

to do

10 Conclusion

This chapter contains in section 10.1 the conclusion and respectively, in section 9.1 the future work.

10.1 Conclusion

The COOP attack which can manifest due to a series of factors such as a memory corruption, layout leackage of the binary and presence of useful gadgets in sufficient large execuatbles is a serious security threat. We have developed *TypeShild*, a runtime fine grained CFI enforcing tool which can precissely filter legitimate from illegitimate indirect forward gadgets in executables. It uses a novel run-time type checking technique based on function parameter type checking and parameter counting in order to efficiently filter-out legitimate and illegitimate forward edges. *TypeShild* provides more precise coverage than existing approaches with smaller performance overhead. We have implemented *TypeShild* and applied it to real software such as: web servers, JavaScript engines, FTP servers and large-scale software including Chrome and Firefox browsers. We demonstrated through extensive experiments and comparisons with related software that *TypeShild* has higher precision and lower performance overhead than the existing state-of-the-art tools.

you can extend this up to a page. If you do than please keep (comment it out) also the old version of the conclusion comment it out

Acronyms

CRA	Code Reuse Attack
BLA	kkk

List of Figures

List of Tables

Bibliography

[1] M. 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] D. Jang et al., 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] C. Tice et al., Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM, In *Proceedings of the USENIX conference on Security (USENIX SEC)*, ACM, 2014.

[5] S. Crane et al., It's a TRaP: Table Rndomization and Protection against Function-Reuse Attacks, In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[6] 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.

[7] Intel, Intel CET, `http://blogs.intel.com/evangelists/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks/`, 2016.

[8] Microsoft, Windows Control Flow Guard, `https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx`, 2015.

[9] 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.

[10] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, H. Bos, ShrinkWrap: VTable Protection Without Loose Ends, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[11] 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.

[12] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner, Protecting C++ Dynamic Dispatch Through VTable Interleaving, In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[13] 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.

[14] 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.

[15] 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.

[16] Juan Caballero, and Zhiqiang Lin, Type Inference on Executables, In *ACM Computing Surveys*, (CSUR), 2016.

[17] V. van der Veen et al., Practical Context-Sensiticve CFI, In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[18] 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.

[19] 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.

[20] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz, BAP: A Binary Analysis Platform, In *Proceedings of Computer Aided Verification (CAV)*, 2011.

[21] Alexander Fokin, Yegor Derevenets, Alexander Chernov, and Katerina Troshina, SmartDec: Approaching C++ decompilation, In *Working Conference on Reverse Engineering (WCRE)*, 2011.

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

[23] Alan Mycroft, Lecture Notes, In *https://www.cl.cam.ac.uk/ am21/papers/sas07slides.pdf*, 2007.

[24] 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.

[25] Dynamic Instrumentation Tool Platform DynamoRIO, In *http://dynamorio.org/home.html*.

[26] N. Carlini et al., Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, In *Proceedings of the USENIX conference on Security (USENIX SEC)*, ACM, 2015.

[27] R. Skowyra et al., Systematic Analysis of Defenses Against Return-Oriented Programming, In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2013.

[28] B. Niu et al., Modular Control-Flow Integrity, In *ACM Conferece on Programming Language Design and Implementation (PLDI)*, 2014.

[29] R. Skowyra et al., Hardware-Assisted Fine-Grained Code-Reuse Attack Detection, In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.

[30] E. Göktaş et al., Out Of Control: Overcoming Control-Flow Integrity, In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[31] M. Abadi et al., Control Flow Integrity, In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[32] B. Niu et al., RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Inegrity, In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[33] B. Niu et al., Per-Input Control-Flow Integrity, In *Proceedings the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[34] M. Abadi et al., Control Flow Integrity Principles, Implementations, and Applications, In *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[35] N. Carlini et al., ROP is still dangerous: Breaking Modern Defenses, In *Proceedings of the USENIX conference on Security (USENIX SEC)*, ACM, 2014.

[36] M. Wang et al., Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[37] A. J. Mashtizadeh et al., CCFI: Cryptograhically Enforced Control Flow Integrity, In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[38] V. Mohan et al., Opaque Control-Flow Integrity, In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[39] C. Zhang et al., VTint: Protecting Virtual Function TablesÍntegrity, In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[40] C. Zhang et al., Practical Control Flow Integrity & Randomization for Binary Executables, In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[41] M. Zhang et al., Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-ROP Attcks, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[42] T. Petsios et al., DynaGuard: Armoring Canary-based Protections against Brute-force Attacks, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[43] A. Prakash et al., Defeating ROP Through Denial of Stack Pivot, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[44] A. Prakash et al., Strict Protection for Virtual Function Calls in COTS C++ Binaries, In *Symposium on Network and Distributed System Security (NDSS)*, 2015.