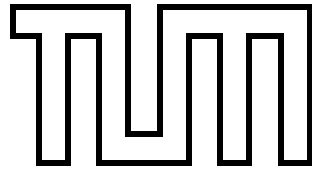


---

---

---





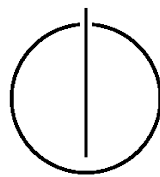
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

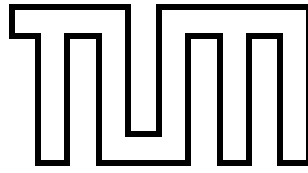
Masterarbeit in Informatik

**to do title eng**

Matthias Konstantin Fischer







# FAKULTÄT FÜR INFORMATIK

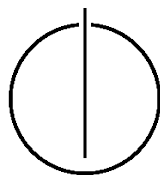
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

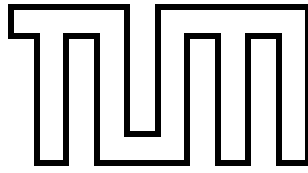
to do title eng

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

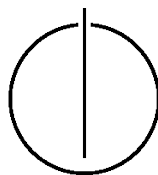
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

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 22. 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 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/caller 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 corruption vulnerabilities. Our evaluation revealed that *TypeShield* imposes up to X% and X% overhead for performance-intensive benchmarks on the Chrome and Firefox browsers, respectively.

---

# Contents

<b>Acknowledgements</b>	<b>ix</b>
<b>Abstract</b>	<b>xi</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Motivation . . . . .	2
1.2. Research Goals . . . . .	2
1.3. Outline . . . . .	2
<b>2. Overview</b>	<b>3</b>
2.1. Types Inference/Recovery from Binaries . . . . .	3
2.1.1. Types Recovery in General . . . . .	3
2.1.2. Function Parameter Types Recovery . . . . .	3
2.1.3. Caller/Callee Parameter Types Recovery . . . . .	3
2.2. Code-Reuse Attacks . . . . .	3
2.3. Control-Flow Integrity . . . . .	3
2.4. Cntrol-Flow Integrity . . . . .	3
<b>I. Snippets</b>	<b>5</b>
<b>3. Snippet [AddressTaken]</b>	<b>7</b>
3.1. step 1 . . . . .	7
3.2. step 2 . . . . .	7
<b>4. Implementation</b>	<b>9</b>
<b>5. Evaluation</b>	<b>11</b>
5.1. R1: Effectiveness of our Tool . . . . .	11
5.2. R2: Precision of our Tool . . . . .	15
5.3. R3: Performance overhead of our Tool . . . . .	15
5.4. R4: Instrumentation overhead of our Tool . . . . .	15
5.5. R5: Security coverage of our tool . . . . .	15
5.6. R6: Which kind of attacks can our tool defend off . . . . .	15
5.7. R7: Whar are the limitations of our Tool . . . . .	15
5.8. R8: To Do. . . . .	15
5.9. Classification . . . . .	15
5.9.1. Callsites . . . . .	15
5.9.2. Calltargets . . . . .	15

5.10. Patching Policies . . . . .	15
5.10.1. AT . . . . .	15
5.10.2. ParamCount . . . . .	15
5.10.3. ParamType . . . . .	16
5.11. Security Evaluation . . . . .	16
5.12. Performance . . . . .	16
<b>6. Discussion</b>	<b>17</b>
6.1. How to make the type inference more precise? . . . . .	17
6.2. Comparison with TypeArmor and why are we better than TypeArmor? . . .	17
6.3. Whys is not TypeArmor working as it should to? . . . . .	17
6.4. What is not clear bout TypeArmor? . . . . .	17
6.5. What can for sure not work as in TypeArmor paper explained? . . . . .	17
<b>7. Related Work</b>	<b>19</b>
7.1. Mitigation of Advanced Code-Reuse Attacks . . . . .	19
7.2. Type-Inference on Executables . . . . .	20
7.3. Binary-based Protection against COOP . . . . .	20
7.4. Source Code based Protection against COOP . . . . .	20
7.5. Runtime-based Protection against COOP . . . . .	21
<b>8. Conclusion</b>	<b>23</b>
8.1. Future Work . . . . .	23
<b>9. Conclusion</b>	<b>25</b>
9.1. Conclusion . . . . .	25
<b>Acronyms</b>	<b>27</b>
<b>Bibliography</b>	<b>33</b>



# 1. Introduction

Control-Flow Integrity (CFI) [32, 29] 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 in 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 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 [9] 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 TypeArmor 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

## 1. Introduction

---

2. We did this

3. We did this.

The rest of the MA is organized as follows.

### 1.1. Motivation

here comes the Motivation.

2-3 sentences.

### 1.2. Research Goals

here comes 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 the technique used in *TypeShild*. Chapter 4 presents briefly the implementation details of our tool. The *TypeShild* implementation is evaluated and discussed in Chapter 5 and Chapter 6, respectively. Chapter 7 surveys related work. Finally, Chapter 8 highlights several future steps and Chapter 9 concludes this thesis.

## **2. Overview**

### **2.1. Types Inference/Recovery from Binaries**

#### **2.1.1. Types Recovery in General**

#### **2.1.2. Function Parameter Types Recovery**

#### **2.1.3. Caller/Callee Parameter Types Recovery**

### **2.2. Code-Reuse Attacks**

this section will up to 1 DIN A page. First talk about code reuse attacks in general and then about COOP.

Say why COOP is not affected by the following mitigation approaches Talk about intel CET, talk about Windows CF guard.

### **2.3. Control-Flow Integrity**

### **2.4. Cntrol-Flow Integrity**



**Part I.**

**Snippets**



### 3. Snippet [AddressTaken]

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 3.1** *!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.

#### 3.1. step 1

We rely on Dyninst to tell us the boundaries of the .plt section !todo! add more information here and the .text section, which contain the executable part of the binary and thus are use to precheck any addresses that we might find in this step.

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).

#### 3.2. step 2

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

### 3. Snippet [AddressTaken]

---

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 ?



## 4. Implementation

In the Master thesis this section should be no longer than 1 DIN A page:

example of implementation text from USENIX cover Paper.

Please write in the same style.

We implemented C A V ER based on the LLVM Compiler project [43] (revision 212782, version 3.5.0). The static instrumentation module is implemented in Clang's CodeGen module and LLVM's Instrumentation module. The runtime library is implemented using the compiler-rt module based on LLVM's Sanitizer code base. In total, C A V ER is implemented in 3,540 lines of C++ code (excluding empty lines and comments). C A V ER is currently implemented for the Linux x86 platform, and there are a few platform-dependent mechanisms. For example, the type and tracing functions for global objects are placed in the .ctors section of ELF. As these platform-dependent features can also be found in other platforms, we believe C A V ER can be ported to other platforms as well. C A V ER interposes threading functions to maintain thread contexts and hold a per-thread red-black tree for stack objects. C A V ER also maintains the top and bottom addresses of stack segments to efficiently check pointer membership on the stack. We also modified the front-end drivers of Clang so that users of C A V ER can easily build and secure their target applications with one extra compilation flag and linker flag, respectively.



## 5. 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 our tool in securing binary programs against the COOP attack?
- **R2:** How precise is our tool in detecting the types of the caller/caller pairs?
- **R3:** What is the performance overhead of our tool?
- **R4:** What are the instrumentation overheads imposed by our tool?
- **R5:** How many caller/called pairs are secured by our tool and how many remain unsecured?
- **R6:** Against which kind of attacks can our tool secure programs?
- **R7:** What are the Limitations of our Tool?
- **R8:** List is not exhaustive. Give another relevant research question. if there is one.

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

**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.

### 5.1. R1: Effectiveness of our Tool

Table 5.1.: 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 5.2.: 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

## 5. Evaluation

---

Table 5.3.: 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 5.4.: 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 5.5.: Calltarget Classification

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

Table 5.6.: Coumpound table

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

Table 5.7.: MAtching table

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

Table 5.8.: 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 5.9.: 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 5.10.: 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 5.11.: 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

alternative for abobe:

Figure 5.1.: impact of CFI and CFC

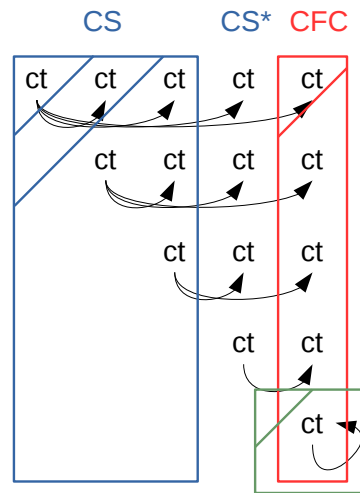


Table 5.12.: matching

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 5.2.: liveness iteration, dummy

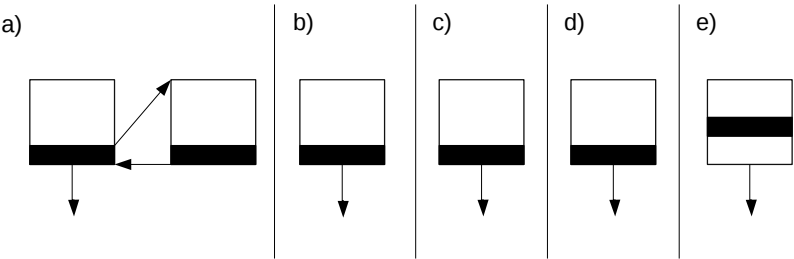


Figure 5.3.: reaching iteration, dummy

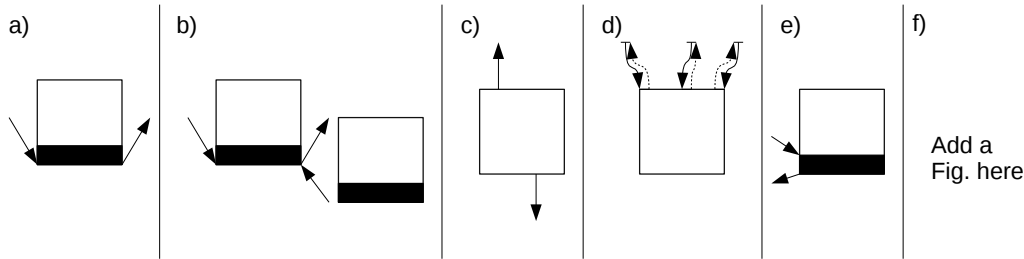


Table 5.13.: pairings compares

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

Table 5.14.: 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

## 5.2. R2: Precision of our Tool

## 5.3. R3: Performance overhead of our Tool

Table 5.15.: 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

## 5.4. R4: Instrumentation overhead of our Tool

## 5.5. R5: Security coverage of our tool

## 5.6. R6: Which kind of attacks can our tool defend off

## 5.7. R7: Whar are the limitations of our Tool

## 5.8. R8: To Do.

it is easier for the reader if we can directly map those section from underneath on the section from above.

## 5.9. Classification

### 5.9.1. Callsites

overestimation param count. table. number of parameters.

### 5.9.2. Calltargets

underestimation param table.

## 5.10. Patching Policies

Two types of diagrams. Table 5 from TypeArmor and a CDF to compare param count and param type. (baseline).

### 5.10.1. AT

### 5.10.2. ParamCount

table, cdf, baseline vs. server. approximations.

### **5.10.3. ParamType**

table, cdf, baseline vs. server. approximations.

## **5.11. Security Evaluation**

## **5.12. Performance**

spec 2006.



## 6. Discussion

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

### 6.1. How to make the type inference more precise?

1/2 page

### 6.2. Comparison with TypeArmor and why are we better than TypeArmor?

1/2 page

### 6.3. Whys is not TypeArmor working as it should to?

1/2 page

### 6.4. What is not clear bout TypeArmor?

1/2 page

### 6.5. What can for sure not work as in TypeArmor paper explained?

Furthermore, bla ...



## 7. Related Work

This chapter we shortly review the main techniques and tools which are related to *Type-Shield*. Section 7.1 presents several techniques which can not fully defend against advanced CRAs. Section 7.2 gives an overview of tools used to recover type inference from binaries. Section 7.3 highlights several tools which can mitigate against the COOP attack on the binary level. Section 7.4 presents tools which can mitigate against the COOP source code level. Finally, section 7.5 shows some of the most promising runtime-based mitigation techniques against COOP.

### 7.1. 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/callee 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 plethora 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 [38], *(c)* coarse-grained CFI based on binary loader, CFCI [39] *(d)* fine-grained code randomization, O-CFI [36], *(e)* cryptography with hardware support, CCFI [35], *(f)* ROP stack pivoting, PBlocker [41], *(g)* canary based protection, DynaGuard [40], *(h)* checking vTable integrity for protecting against COOP based on CFI for source code such as SafeDispatch [2], vtv [4] LLVM and GCC compiler based vor vTable protection and binary rewriting such as vfGuard [42], vTint [37] and [5], *(i)* runtime and hardware support based on a combination of LBR, PMU and BTS registers CFIGuard [27] and *(j)* source code recompilation with CFI and/or randomization enforcement against JIT-ROP attacks, MCFI [26], RockJIT [30] and PiCFI [31].

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

### 7.2. 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. *TypeShield* is based on DynInst and the quality of the executable disassembly fits our needs. For a more comprehensive review on the capabilities of DynInst and other tools we advise the reader to have a look at [18].

Second, alias analysis in binaries 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 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. [16]. Interesting to notice is that the code from only a few of these tools is available.

*TypeShield* 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

### 7.3. Binary-based Protection against COOP

*TypeShield* is most similar to TypeArmor [9] 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.

### 7.4. Source Code based Protection against COOP

There are several source code based tools which can successfully 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 through 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.

## 7.5. Runtime-based Protection against COOP

“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.

Intel CET [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 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 [8] 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 [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 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 tripped by adding legitimate indirect call sites during the COOP attack.



## 8. Conclusion

This chapter presents in section 8.1 the future steps in order to improve the precision and efficiency of *TypeShild*.

### 8.1. 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





## 9. Conclusion

This chapter contains in section 9.1 the conclusion and respectively, in section 8.1 the future work.

### 9.1. Conclusion

The COOP attack which can manifest due to a series of factors such as a memory corruption, layout leakage of the binary and presence of useful gadgets in sufficient large executables is a serious security threat. We have developed *TypeShield*, a runtime fine grained CFI enforcing tool which can precisely 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. *TypeShield* provides more precise coverage than existing approaches with smaller performance overhead. We have implemented *TypeShield* 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 *TypeShield* 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 then please keep (comment it out) also the old version of the conclusion comment it out



# Acronyms

<b>CRA</b>	Code Reuse Attack
<b>BLA</b>	kkk



# List of Figures

5.1.	impact of CFI and CFC . . . . .	13
5.2.	liveness iteration, dummy . . . . .	14
5.3.	reaching iteration, dummy . . . . .	14



## List of Tables

5.1. Classification CS . . . . .	11
5.2. Compound . . . . .	11
5.3. Classification CT . . . . .	12
5.4. Callsite Classification for paramcount . . . . .	12
5.5. Calltarget Classification . . . . .	12
5.6. Coumpound table . . . . .	12
5.7. MAtching table . . . . .	12
5.8. policy evaluation . . . . .	12
5.9. param wideness . . . . .	12
5.10. tabelle 7 . . . . .	13
5.11. tabelle 7 . . . . .	13
5.12. matching . . . . .	13
5.13. pairings compares . . . . .	14
5.14. policy baseline . . . . .	14
5.15. Performance Tabble, more better is a bar chart here! . . . . .	15





# 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](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 Payert, 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-Sensitive 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] 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.

- [25] 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.
- [26] B. Niu et al., Modular Control-Flow Integrity, In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.
- [27] 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.
- [28] E. Göktaş et al., Out Of Control: Overcoming Control-Flow Integrity, In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [29] M. Abadi et al., Control Flow Integrity, In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.
- [30] B. Niu et al., RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity, In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.
- [31] B. Niu et al., Per-Input Control-Flow Integrity, In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.
- [32] M. Abadi et al., Control Flow Integrity Principles, Implementations, and Applications, In *ACM Transactions on Information and System Security (TISSEC)*, 2009.
- [33] N. Carlini et al., ROP is still dangerous: Breaking Modern Defenses, In *Proceedings of the USENIX conference on Security (USENIX SEC)*, ACM, 2014.
- [34] M. Wang et al., Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.
- [35] A. J. Mashtizadeh et al., CCFI: Cryptographically Enforced Control Flow Integrity, In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
- [36] V. Mohan et al., Opaque Control-Flow Integrity, In *Symposium on Network and Distributed System Security (NDSS)*, 2015.
- [37] C. Zhang et al., VTint: Protecting Virtual Function Tables Integrity, In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.
- [38] 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.
- [39] M. Zhang et al., Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-ROP Attacks, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.
- [40] T. Petsios et al., DynaGuard: Armoring Canary-based Protections against Brute-force Attacks, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

- [41] A. Prakash et al., Defeating ROP Through Denial of Stack Pivot, In *Annual Computer Security Applications Conference (ACSAC)*, 2015.
- [42] A. Prakash et al., Strict Protection for Virtual Function Calls in COTS C++ Binaries, In *Symposium on Network and Distributed System Security (NDSS)*, 2015.