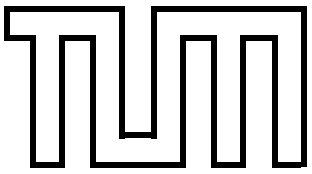


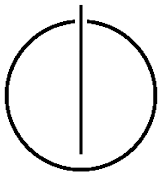
FAKULTÄT FÜR INFORMATIK

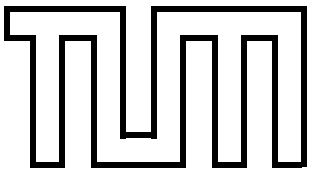
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

to do title eng

Matthias Konstantin Fischer





FAKULTÄT FÜR INFORMATIK

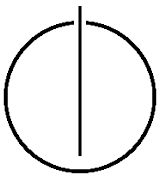
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

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



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

München, den 4. November 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) [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 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 [8] 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: 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 devel- opment significantly. Nevertheless, a dedicated attacker is still able to achieve code execution [33, 25]. Information leaks are utilized to counter ASLR and reveal the lay- out 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 ar- bitrary 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 at- tacks 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 high- light the limitations of the proposed defense methods. Although single implementa- tions 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 loose their simplicity by becom- ing 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 gad- gets.

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 constrain 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 *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. C++ Forbidden Calls Exposed

This Chapter presents in Section 2.1 a brief overview about polimorphysm 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 those vTable entries which for a single indirect call site lie in the vTable hier-

```
1  class nsMultiplexInputStream final
2   :public nsIMultiplexInputStream
3   ,public nsISeekableStream //A1
4   ,public nsIIPCSerializableInputStream //A2
5   ,public nsICloneableInputStream{ //A3
6  nsTArray<nsCOMPtr<nsIInputStream>> mStreams;
7  NS_IMETHODIMP nsMultiplexInputStream::Close(){
8    MutexAutoLock lock(mLock);
9    mStatus = NS_BASE_STREAM_CLOSED;
10   //set NS_OK flag
11   nsresult rv = NS_OK;
12   //get array length
13   uint32_t len = mStreams.Length();
14   //array-based main loop gadget
15  for (uint32_t i = 0; i<len; ++i){
16   //(1) hijacked indirect call
17   nsresult rv2=mStreams[i]->Close();
18   if (NS_FAILED(rv2)) {
19       rv = rv2;
20   }
21  }
22   return rv;
23 }
```

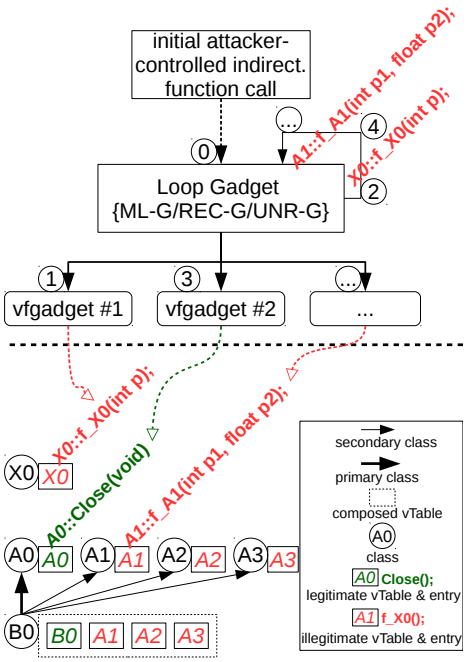


Figure 2.1.: Code example used to illustrate how a COOP loop gadget works

archy. 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 exploiter 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-rime 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 inheritance paths. vTrust [12] (average runtime overhead 2.2%) enforces two lay-

ers 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/caller 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 addresses 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 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 than the attacker could highjack 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 applica-

tion 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 number of parameters and return types. The final goal of this attack is to prepare

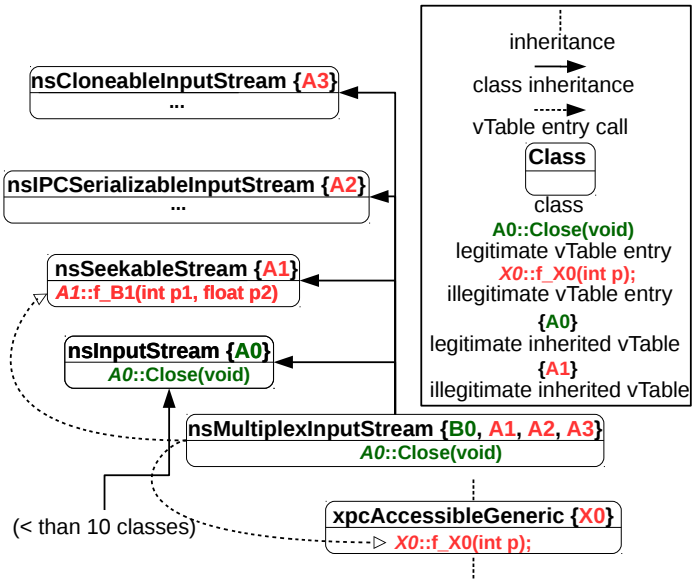


Figure 2.2.: Class inheritance hierarchy of the classes involved in the COOP attack against the Firefox browser. Red letters indicate forbidden vTble entries and green letters indicate allowed vTable entries for the given indirect call site contained in the main loop gadget.

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. 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 !todo! name based on the di-opt environment from patharmor [8], which relies on the Dyninst library [22] (updated to version 9.2.0). We provided our own pass to analyze and patch given binaries, which consists of !todo! measure locs lines of C++ code. The core part of our pass is an instruction analyzer, which relies on the DynamoRIO library [23] (version 6.6.1) to decode instructions an provide information. This analyser is then used to implement an address taken analysis and configurable versions of the backward reaching definitions algorithm and the forward liveness analysis algorithm, which are both based on the algorithms presented in [8]. Callsites and calltargets are then analyzed using those two algorithms, which are configured based on the origin and the chosen policy, which is decided during compilation time.

Additionally to measure the quality of our tool, we wrote a `MachineFunction` pass for llvm/clang version 4.0.0 (trunk 283889) in the x86 target code generation portion, to have the lowest possible difference between the binary and the IR of llvm. This pass provides us with ground truth data regarding callsites and calltargets, which we then use in our python evaluation and test environment. We are able to measure various metrics using this environment, which we present in this work.

Our current implementation allows for analysis and patching of any native elf binary on linux x86-64.

We implemented *TypeShild* based on the DynInst [22] 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 [23]. 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:

Figure 6.1.: impact of CFI and CFC

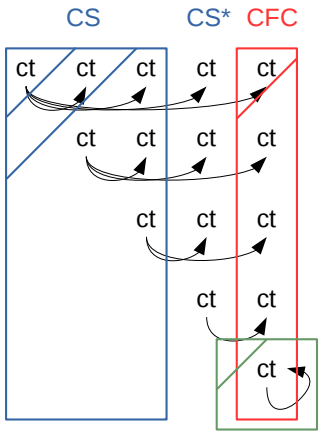


Figure 6.2.: liveness iteration, dummy

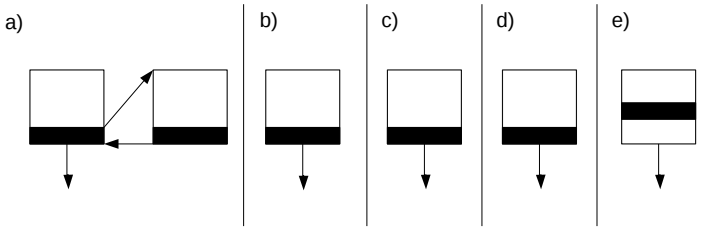


Figure 6.3.: reaching iteration, dummy

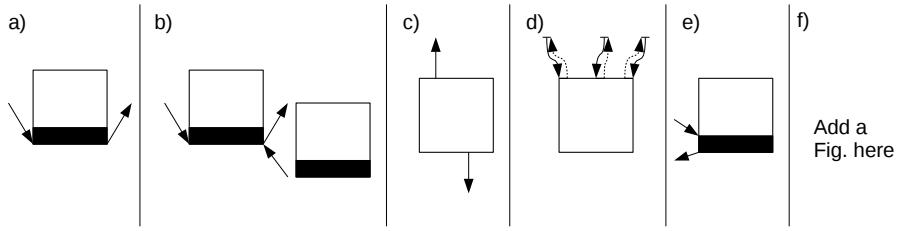


Table 6.13.: matching

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

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

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
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 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. 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 [16].

Second, alias analysis in binares 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 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. [14]. 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

8.2. 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)* cryptografy 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/calle 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/VTV [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 succesfully protect against the COOP attack. Such tools are: ShrinkWrap [9], IFCC/VTV [4], SafeDispatch [2], vTrust [12], Readactor++ [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 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.

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.

TypeShild is most similar to TypeArmor [8] 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.

Runtime based: "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 [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 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 [7] 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 [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 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

Part I.

SNIPPETS

11. Snippet [COUNT policy]

What we call the COUNT policy is essentially the policy introduced by typearmor[**?**]. The basic idea revolves aroung classifying calltargets by the number of parameters they provide and callsites by the number of paramters they require. 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 SystemV ABI designates as parameter registers and completely ignores other registers like floating point ones. 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. The main problem however, is that while their 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:

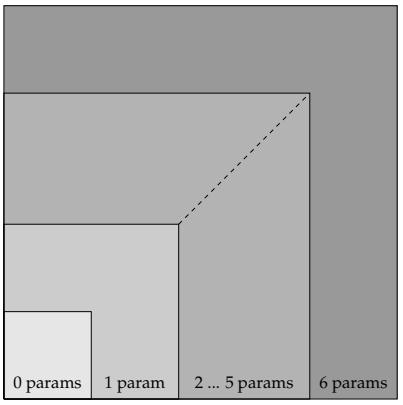


Figure 11.1.: COUNT policy schema

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

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.

To aquire the classification of calltargets, a forward liveness analysis is used, which works rather well with 0 overestimations in 6 of 8 testargets and 1 overestimation in mysqld and postgres and perfect classification percentage between 67% and 88%. However the reaching analysis, which is used to classify

callsites is more problematic with only 0 targets showing zero underestimations and all others ranging between 5% and 17%.

12. Snippet [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. In general, these 64bit registers can be accessed in 4 different ways, one can access the whole 64bit register, the lower 32bit part, the lower 16bit part, or the lower 8bit part. Four general purpose registers can also directly access the higher 8bit part of the 16bit lower part, but for our purpose we also registers those as a 16bit access. Based on this information, we can assign a register one of 5 possible simplistic types $\mathcal{T} = 64, 32, 16, 8, 0$. We also included the type 0 to model the absence of data within a register. Now similar to typearmor we also allow overestimation of types in callsites and underestimation of types in calltargets. However the matching idea is different, because as can we depict in Figure 12.1, 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 fulfillst 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 the 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.

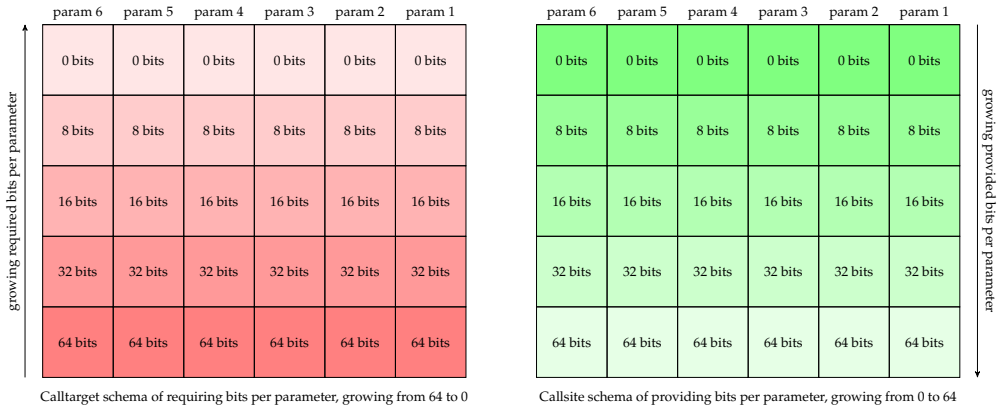


Figure 12.1.: TYPE policy schema for callsites and calltargets

13. Snippet [Matching]

To verify our classification of callsites and calltargets and measure the reduction in calltargets per callsite, we need to compare the data produced by our tool to some sort of ground truth. We acquire the latter from a clang/llvm machine function pass, which allows us to collect the expected return type and provided parameters for callsites and the provided return type and expected paramter counts for calltargets. To be able to compare those two datasets, we need to be able to structurally match them.

First, we look at calltargets, we exclude internal functions, that are provided by any C or C++ binary and match based on function names. While C programs employ simple names, C++ programs have typed names due to the possibility of function overloading and therefore use a mangling scheme, requiring us to demangle the names from both sets and compensate any differences from the different sources. Table **??** shows three data points regarding calltargets: First the number of calltargets that can be found in both datasets and are therefore comparable; Second, the number of calltargets that are found by our tool but not by our clang/llvm pass, which are problematic, because we cannot get data regarding their classification quality, the only real noteworth is the node test target with 339 calltargets not found in the pass dataset, however considering it is only about 1.6% of all calltargets, we can essentially ignore this number; Third, the number of calltargets that are found by our clang/llvm pass but not by our tool, which is mostly due to the fact that we collect our data from the whole compilation and with our current setup cannot differentiate between different build targets and therefore this number is of no real concern to us.

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. The are several reasons for mismatch: On 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.

Target	calltargets			callsites		
	match	clang miss	padyn miss	match	clang miss	padyn miss
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%)	6497	1507 (18.82%)	1070 (14.14%)
postgres	6930	1 (0.01%)	1512 (17.91%)	526	152 (22.41%)	105 (16.64%)
memcached	133	0 (0.0%)	91 (40.62%)	3	94 (96.9%)	45 (93.75%)
node	20638	339 (1.61%)	620 (2.91%)	8599	1983 (18.73%)	1894 (18.05%)

Table 13.1.: Table shows the quality of structural matching provided by our automated verify and test environment, regarding callsites and calltargets. Missmatches are basically the result of discrepancies between llvm/IR and the actual binary

14. Snippet [Theoretical Limits]

short presentation regarding the theoretical limits of the COUNT and the TYPE policy and 1page (including table)

15. Snippet [Patching Schema]

presenting the actual patching schema including diagrams 1 - 2 pages (possibly 3) describing the conceptual basics of the dyninst relocation schema to explain the patching process some short ASM explanation

16. Snippet [Baseline COUNT Implementation]

Will take about 5 pages, probably 1/2 page per table

Target	#CS	-x	+0	+1	+2	+3	+4	+5	+6
proftpd	112	4	100	0	6	2	0	0	0
vsftpd	14	0	14	0	0	0	0	0	0
lighttpd	48	8	34	0	0	1	0	5	0
nginx	234	28	181	10	11	2	0	2	0
mysqld	6497	667	4628	220	132	234	224	392	0
postgres	526	30	452	8	10	14	2	10	0
memcached	3	0	3	0	0	0	0	0	0
node	8599	493	5947	546	257	830	314	212	0

Table 16.1.: Table shows the overestimation of the parameter count in matched callsites that is happening in our implementation of the basline implementation of the COUNT policy, with -x denoting problematic callsites

Target	#CT	+x	-0	-1	-2	-3	-4	-5	-6
proftpd	1015	0	874	88	35	7	10	0	1
vsftpd	319	0	232	69	13	3	1	0	1
lighttpd	290	0	251	33	4	2	0	0	0
nginx	921	0	691	192	34	4	0	0	0
mysqld	9934	1	6667	2459	517	194	45	22	29
postgres	6941	1	5762	884	167	84	28	3	12
memcached	133	0	117	11	2	2	0	1	0
node	20800	0	15212	3649	1143	510	138	84	64

Table 16.2.: Table shows the underestimation of the parameter count in matched calltargets that is happening in our implementation of the basline implementation of the COUNT policy, with -x denoting problematic calltargets

Target	**callsites** (param. class.) #	perfect	problem	**calltargets** (param. class.) #	perfect	problem
proftpd	112	100(89.28%)	4(3.57%)	1015	874(86.1%)	0(0.0%)
vsftpd	14	14(100.0%)	0(0.0%)	319	232(72.72%)	0(0.0%)
lighttpd	48	34(70.83%)	8(16.66%)	290	251(86.55%)	0(0.0%)
nginx	234	181(77.35%)	28(11.96%)	921	691(75.02%)	0(0.0%)
mysqld	6497	4628(71.23%)	667(10.26%)	9934	6667(67.11%)	1(0.01%)
postgres	526	452(85.93%)	30(5.7%)	6941	5762(83.01%)	1(0.01%)
memcached	3	3(100.0%)	0(0.0%)	133	117(87.96%)	0(0.0%)
node	8599	5947(69.15%)	493(5.73%)	20800	15212(73.13%)	0(0.0%)

Table 16.3.: Table shows the ability of inferring the parameter count for callsites and calltargets exhibited by our implementation of the COUNT policy.

Target	**callsites** (non-void class.) #	perfect	problem	**calltargets** (void class.) #	perfect	problem
proftpd	112	101(90.17%)	1(0.89%)	1015	3.57(55.96%)	411(40.49)%)
vsftpd	14	14(100.0%)	0(0.0%)	319	0.0(64.57%)	57(17.86)%)
lighttpd	48	44(91.66%)	1(2.08%)	290	16.66(72.75%)	70(24.13)%)
nginx	234	215(91.88%)	0(0.0%)	921	11.96(66.34%)	284(30.83)%)
mysqld	6497	5310(81.73%)	43(0.66%)	9934	10.26(76.02%)	1823(18.35)%)
postgres	526	497(94.48%)	1(0.19%)	6941	5.7(72.32%)	1624(23.39)%)
memcached	3	3(100.0%)	0(0.0%)	133	0.0(76.69%)	19(14.28)%)
node	8599	7060(82.1%)	116(1.34%)	20800	5.73(74.17%)	4185(20.12)%)

Table 16.4.: Table shows the ability of inferring the voidness for callsites and calltargets exhibited by our implementation of the COUNT policy.

17. Snippet [AddressTaken]

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[**?**], which introduced various types of taken addresses. An address is considered to be taken, when it is loaded into memory or a register. It should be noted that this restriction is orthogonal to the actual algorithm and therefore can be disabled, should the restriction cause problems. We were however able to reduce the set of possible calltargets to 18% - 62% of their previous size, in three cases we were able to reduce the number to below 10%. This of course directly translated into a significant impact on the theoretical limits of the type and count base policies with similar numbers.

17.1. Analysis

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

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

!shorthand! 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.

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.

In the first step, we look at three different data sections of the binary, which could possibly contain taken addresses: the .data, .rodata and .dynsym sections. As [**?**] proposed, we slide a four and an eight byte window over the data within those sections and look for addresses that point to function entry blocks. In case of shared libraries, we need to let Dyninst translate the raw address, we exctracted, so we can perform the function check.

In the second step 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.

17.2. Evaluation

To test our implementation we employed the same technique as of before. We generate the ground truth set of all address taken function $\mathcal{F}_{AT}^{clang}$ using a clang/llvm analysis-pass that collects the necessary information during compilation. The set of address taken that we collected using our tool is $\mathcal{F}_{AT}^{padyn}$. As previously, we restrict both sets to the set of functions, that exist in both datasets, to ensure comparability [1]. With this basis, we can generate three sets:

1. !shorthand! address taken (AT) functions existing in both data sets $\mathcal{F}_{AT}^{both} = \mathcal{F}_{AT}^{clang} \cap \mathcal{F}_{AT}^{padyn}$

2. !shorthand! address taken (AT) functions existing in padyn but not in clang $\mathcal{F}_{AT}^{over} = \mathcal{F}_{AT}^{clang} \cap \mathcal{F}_{AT}^{clang}$, which are non problematic, as they simply indicate an overestimation.

3. !shorthand! address taken (AT) functions existing in clang but not padyn $\mathcal{F}_{AT}^{problem} = \mathcal{F}_{AT}^{clang} \cap \mathcal{F}_{AT}^{clang}$, which are possibly problematic, as they might be required functions. However, this should be inspected on an individual basis, as clang/llvm itself also overestimates the set of !shorthand! address taken (AT) functions.

[1] We are restricted to names to match functions, thus we usually cannot match plt functions, because most of the time their names are stripped away after compilation and dyninst then reports these targ$address

Target	$\mathcal{F}_{AT}^{both}$	$\mathcal{F}_{AT}^{over}$	$\mathcal{F}_{AT}^{problem}$
proftpd	388	2 (0.51%)	3 (0.76%)
vsftpd	9	1 (10.0%)	0 (0.0%)
lighttpd	55	4 (6.77%)	0 (0.0%)
nginx	479	64 (11.78%)	0 (0.0%)
mysqld	5747	132 (2.24%)	363 (5.94%)
postgres	562	1928 (77.42%)	4 (0.7%)
memcached	11	3 (21.42%)	0 (0.0%)
node	7005	522 (6.93%)	420 (5.65%)

Table 17.1.: Table that shows how much our address taken analysis differs from the ground truth provided by our clang/llvm passs

As we can see in table 17.1, most of our test targets are within reasonable bounds regarding problematic and overestimation. The postgres target has a much higher percentage in overestimation than the rest, which is not problematic, as it only lowers our possible calltarget reduction effect. The two targets mysqld and node might be of more concern to us, as we probably cannot fully attribute the difference to the overestimation in llvm/clang. In these two cases one needs to evaluate, whether our version of AT analysis is applicable by rigorous testing of the planned usecases, or resort to a more conservative strategy.

+ 1/2 page for effect on theoretical limit how to reduce the table ?

Target	Policy	unfiltered CTs	clang/llvm filtered CTs	filtered CTs
proftpd	AT	1015± 0	391± 0	390± 0
proftpd	COUNT*	860.93 ± 174.9	349.96 ± 59.56	353.64 ± 60.97
proftpd	TYPE*	727.95 ± 129.0	334.59 ± 55.44	332.65 ± 55.14
vsftpd	AT	318± 0	9± 0	10± 0
vsftpd	COUNT*	213.85 ± 58.27	7.85 ± 1.35	8.14 ± 1.8
vsftpd	TYPE*	173.71 ± 62.79	5.0 ± 3.16	5.28 ± 3.61
lighttpd	AT	290± 0	55± 0	59± 0
lighttpd	COUNT*	215.91 ± 83.03	36.51 ± 14.8	38.43 ± 17.17
lighttpd	TYPE*	180.64 ± 66.89	30.06 ± 9.68	30.56 ± 10.7
nginx	AT	921± 0	479± 0	543± 0
nginx	COUNT*	520.19 ± 243.08	279.39 ± 130.14	313.87 ± 150.2
nginx	TYPE*	501.34 ± 234.86	277.09 ± 129.69	311.06 ± 149.36
mysqld	AT	9742± 0	6110± 0	5879± 0
mysqld	COUNT*	8715.96 ± 2827.94	6383.99 ± 1705.05	6309.84 ± 1681.68
mysqld	TYPE*	8041.52 ± 2118.28	6083.39 ± 1352.68	5995.2 ± 1325.29
postgres	AT	6930± 0	566± 0	2490± 0
postgres	COUNT*	4743.89 ± 1752.2	388.21 ± 186.32	2092.77 ± 714.8
postgres	TYPE*	3867.06 ± 1293.56	301.34 ± 138.07	1973.52 ± 662.27
memcached	AT	133± 0	11± 0	14± 0
memcached	COUNT*	203.0 ± 0.0	12.0 ± 0.0	21.0 ± 0.0
memcached	TYPE*	160.0 ± 0.0	10.0 ± 0.0	14.0 ± 0.0
node	AT	20638± 0	7425± 0	7527± 0
node	COUNT*	13626.22 ± 4610.72	5628.49 ± 1613.25	5705.04 ± 1649.92
node	TYPE*	12250.84 ± 3501.64	5216.01 ± 1237.21	5279.89 ± 1269.09

Table 17.2.: Table shows the effect of the various AT filtering versions on the policies AT, COUNT* (theoretical limit of COUNT policy) and TYPE* (theoretical limit of TYPE policy) for all of our test targets

Acronyms

CRA	Code Reuse Attack
BLA	kkk

Contents

List of Figures

List of Tables

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`, 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 oaravindf 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-Sensiticve 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 AidedVTint: Protecting Virtual Function Tab 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 executablesv, 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 Conferece on Programming Language Design and Implementation (PLDI)*, 2014.

[29] Ben Niu, and Gang Tan, "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Inegrity", 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 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 TablesÍntegrity", 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 Attcks", 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: Cryptograhically 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.