RUHR-UNIVERSITÄT BOCHUM

Horst Görtz Institute for IT Security

**Technical Report TR-HGI-2014-004**

# Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs

*Robert Gawlik and Thorsten Holz*

Chair for Systems Security

Ruhr-Universität Bochum
Horst Görtz Institute for IT Security
D-44780 Bochum, Germany

TR-HGI-2014-004
December 9, 2014

RUHR UNIVERSITÄT BOCHUM RUB

hgi Horst Görtz Institut für IT-Sicherheit

# Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs

Robert Gawlik and Thorsten Holz

December 9, 2014

**Abstract**

Web browsers are one of the most used, complex and popular software systems nowadays. They are prone to *use-after-free* vulnerabilites and this is the de-facto way to exploit them. From a technical point of view, an attacker uses a technique called *vtable hijacking* to exploit such bugs. More specifically, she crafts bogus virtual tables and lets a freed C++ object point to it in order to gain control over the program at virtual function call sites.

In this paper, we present a novel approach towards mitigating and detecting such attacks against C++ binary code. We propose a static binary analysis technique to extract virtual function call site information in an automated way. Leveraging this information, we instrument the given binary executable and add runtime policy enforcements to thwart the illegal usage of these call sites. We implemented the proposed techniques in a prototype called T-VIP and successfully hardened three versions of Microsoft's Internet Explorer and Mozilla Firefox. An evaluation with several zero-day exploits demonstrates that our method prevents all of them. Performance benchmarks both on micro and macro level indicate that the overhead is reasonable with about 2.2%, which is slightly higher compared to recent compiler-based approaches that address this problem.

## 1 Introduction

Within the last decade, we observed a shift in how attackers compromise systems. Instead of targeting server applications such as common Windows network services, adversaries nowadays often launch attacks against client applications. Especially office applications and browsers (including their plugins) are affected, mainly since they are complex software products and thus prone to software defects. Modern web browsers are highly complex programs: they are capable of numerous tasks including media playback and manipulation, 3D gaming, interpretation of several computer languages, and real-time communication. Furthermore, browsers can serve as the basis for complete usermode environments in operating systems (e.g., Chrome OS [41] or FirefoxOS). Most modern web browsers are developed in C and C++ and consists of several million lines of code. These languages are prone to programming mistakes, which can result in exploitable vulnerabilities hard to spot in such huge programs (both on the source code and on the binary level).

**Vtable Hijacking Attacks** Particular kinds of programming mistakes that are prevalent today result in so called *use-after-free* vulnerabilities. These *temporal* safety problems are often abused by adversaries [2]. In a use-after-free bug, a program path exists during which a pointer to an object that was previously freed, is used again. This dangling pointer could cause the program to crash, unexpected values could be used, or an adversary could even execute arbitrary code.

Especially use-after-free software defects are frequently used during targeted attacks. For example, government agencies recently used a zero-day exploit that takes advantages of such vulnerabilities against Mozilla Firefox to identify users of the TOR network to deanonymize suspects [37]. During the yearly *pwn2own* competition, security researchers showed several times that

such vulnerabilities are prevalent in browsers and successfully demonstrated exploits against Internet Explorer, Mozilla Firefox and Google Chrome. Furthermore, several recent zero-day exploits for Microsoft's Internet Explorer were based on such use-after-free vulnerabilities [33]. In fact, a recent study suggests that 69% of all vulnerabilities in browsers and 21% of all vulnerabilities in operating systems are related to such bugs [14].

To take advantage of such vulnerabilities in object-oriented code, attackers typically utilize a technique called *vtable hijacking*. Compared to traditional attacks like stack-based buffer overflows or formatstring attacks, this technique targets heap-based pointers to virtual tables (shortened: *vtables*), a feature of object-oriented languages like C++. Polymorphic C++ classes have vtables that contain function pointers to the implementation of its virtual methods. If an object of a polymorphic class is freed, but a reference is kept, bogus vtables can be utilized to gain control of the program's control flow. Therefore a bogus vtable is crafted and a pointer to it is injected at memory where the object is pointing to. A virtual function call site is then abused to use the bogus vtable, enabling an adversary to hijack the control flow.

**Preventing Vtable Hijacking** Since vtable hijacking attacks are a frequent problem in practice, several compilers recently started to include protection techniques during the compilation phase. Both GCC and Microsoft Visual Studio started to implement defense solutions. More specifically, GCC introduced the `-fvtable-verify` option [56] that analyzes the class hierarchy during the compilation phase to determine all vtables. Using this information, all virtual function call sites are modified such that virtual method dispatches can be checked during runtime. Similarly, SafeDispatch [25] – as a LLVM extension – inserts checks during compilation phases. VTGuard by Microsoft adds a guard entry at the end of the vtable such that (certain kinds of) vtable hijacks can be detected. Note that these approaches are only applicable to source code since they are implemented during the compilation and link phase. This prevents an adoption to COTS applications where only the binary code is available. However, especially such applications are vulnerable to vtable hijacks.

**Our Approach** In this paper, we present a lightweight approach to provide vtable integrity for COTS binaries implemented in C++ code. We perform our analysis on the binary level since we aim to protect executables for which we do not have source code, debugging symbols, or runtime type information, such as for example web browsers or office applications for Windows. To this end, we introduce a generic method to identify virtual call sites in C++ binary code. More specifically, we lift the assembler code to an *intermediate language* (IL) and then perform backward slicing on the IL level such that we can spot different kinds of C++ virtual function dispatches. In a second step, we instrument each virtual call site and add integrity checks. To this end, we implemented a generic, static binary rewriting engine for PE files that enables us to implement an integrity policy $\mathscr{P}$ for each call site. For now, we have implemented different kinds of integrity policies that, for example, check if a vtable pointer points to a writable memory page (which indicates that an integrity violation happened) or check if a random chosen vtable entry actually *is* a code pointer.

We have implemented our approach in a tool called T-VIP (*towards Vtable Integrity Protection*) that consists of a slicer called vExtractor and a binary rewriting engine called PeBouncer. Experimental results demonstrate that the precision is reasonable and the performance overhead small. Furthermore, our tool was able to mitigate all tested zero-day attacks against Microsoft's Internet Explorer and Mozilla Firefox.

**Contributions** Our main contributions are:

- We introduce an automated method to identify virtual function call sites in C++ binary applications based on an intermediate language and backward slicing. This enables us to determine the potential attack surface for use-after-free and related vulnerabilities in binary executables implemented in C++.

- We present a generic binary rewriting framework for PE executables with low overhead called PeBouncer that we utilize to implement integrity policies for virtual call sites.
- To the best of our knowledge, we are the first to present virtual table integrity protection for binary C++ code *without* the need for source code, debugging symbols, or runtime type information.
- We show that T-VIP protects against sophisticated, real-world use-after-free remote code execution exploits launched against web browsers, including zero-day exploits against Microsoft's Internet Explorer and Mozilla Firefox. A performance evaluation against GCC's virtual table verification feature [56] with micro- and macro-benchmarks demonstrates that our approach introduces a comparable performance overhead.

# 2 Technical Background

Before presenting our approach to enforce the integrity of virtual call sites, we first review the necessary technical background to understand the rest of the paper. More specifically, we explain C++ inheritance and polymorphism and show their manifestation on the internal low-level assembly stage. Furthermore, we discuss how virtual function tables are typically implemented, how this enables use-after-free memory corruption vulnerabilities, and explain why we need an intermediate language to perform our analysis.

## 2.1 C++ Inheritance and Polymorphism

*Inheritance* is a general concept in *Object Oriented Programming* (OOP) languages. Data structures called *classes* can contain data *attributes* and functions named *methods*. Working with classes is mostly done on their instances, which are referred to as *objects*. Classes can serve as base classes when they are inherited, creating derived classes, which inherit the base's attributes and methods in addition to their own attributes and functions. Classes can inherit from multiple base classes, and also, derived classes themselves can serve as base classes, such that a (potentially very complex) class hierarchy is created between them.

A programmer can change the functions of base classes inside derived classes by overloading or implementing them. They must be declared as *virtual*, and any class containing virtual functions is a *polymorphic* class (see Figure 1, left). The binding of virtual functions to a class' instance is performed dynamically during runtime if the compiler cannot determine the instance's type statically. Thereby, the function acts as a message and the instance as the message's receiver. Depending on the dynamically determined type of the instance, the appropriate function is selected, hence, the message has different impacts on the instance according to its runtime type. For more details, the reader is referred to the literature [7, 53].

When compiling C++ code that contains virtual function dispatches, most compilers generate machine code instructions containing indirect calls. These are good constructs for attackers to gain control of the instruction pointer by controlling the call's target register. We elaborate on this danger in the following sections.

## 2.2 Virtual Function Calls

For each class that defines virtual functions, a *virtual function table* (abbr. *vtable*) will be created during compile time. It contains the addresses of all virtual functions that a class provides. During runtime, when an instance is created, a pointer to a vtable is inserted into the instance's layout similar to a data attribute. A class instance's lifetime can involve the usage of several vtables depending on the number of base classes with virtual functions it inherits from.

Figure 1 shows the low level instructions on two virtual function dispatches. At first, a vtable address is loaded into a register (①). Then an indexing offset is added to it to let the register point to the address of the virtual function (②). This is omitted if the virtual function is the vtable's

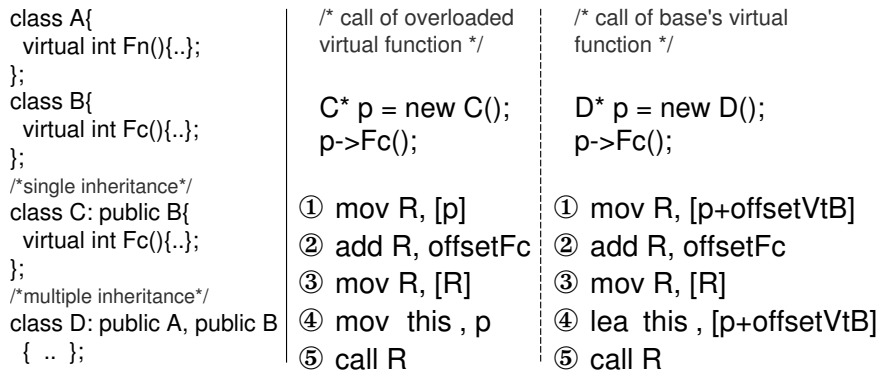


**Figure 1:** Single and multiple inheritance with polymorphic classes (left). C++ and assembly code of dispatching an overloaded virtual function (middle). And an inherited base class' virtual function dispatch (right). Registers are denoted with $R$.

first entry. Afterwards, it is selected by dereferencing the vtable's entry (③) and dispatched with an indirect call (⑤). Additionally, a *this* pointer is created and passed as parameter to the virtual function (④), either via the register `ecx` [35], or as the first parameter. As first parameter, location or registers are used, which are specified in the corresponding calling convention, i.e., the stack or the register `rdi`. The *this* pointer constitutes the instance's address and is adjusted in case of multiple inheritance.

These semantic steps can then be generalized: let $obj$ be the address of an instance and $i$ the displacement offset to the vtable pointer at $obj$. The length of the vtable in bytes is indicated as $|vtable|$. Then, on a 32-bit system, $j \in [0, \frac{|vtable|}{4} - 1]$ denotes the index into the vtable, where an address of a virtual function $vf$ resides. A memory dereference is stated with $mem$. Thus, we get:

$$\forall\, vf\ \exists\, mem\ :\ mem(mem(obj + i) + j * 4) = vf \tag{1}$$

$$\forall\, this\ \exists\, obj\ :\ (obj + i) = this \tag{2}$$

(1) and (2) holds for virtual functions called indirectly, where (1) comprises steps ① - ③ and ⑤, and (2) describes step ④.

Compilers usually translate calls into these five steps [18]. Highly optimized code, such as modern web browser libraries, can omit step ④, combine several steps into single instructions, and have multiple unrelated instructions in between.

There are syntactical varieties in steps ① - ③ and ⑤ dependent on optimization levels. However, the manifestation of semantic step ④ into assembly strongly depends on the used compiler and is independent of the optimization (see Table 1). These subtleties were observed in our 32-bit test binaries originating from C++ code with virtual, single, and multiple inheritance and polymorphic classes, as well as in COTS browser code.

While the syntax may differ, virtual function dispatches reveal themselves in generalized semantics, at least in binary code stemming from GCC, LLVM, and MS Visual C++. As GCC and Visual C++ are standard compilers for browsers on MS Windows, our framework is able to extract virtual function dispatches from their generated code (see Section 4.1).

We refer to the low-level assembly semantics of a C++ virtual function call as *virtual dispatch*, which includes vtable loading, virtual function selection, and passing the *this* pointer as hidden or first parameter to the virtual function. The assembly instruction which performs the indirect call of the virtual function we refer to as *virtual call*.

Recent in-the-wild exploits, including two targeted zero-day attacks against Internet Explorer [33] and one against the Mozilla Firefox version included in the Tor Browser Bundle [37], achieved remote code execution by abusing virtual dispatches. Figure 2 illustrates the different manifestation of the five semantic steps for the three virtual dispatches, utilized to exploit CVE-2013-3897, CVE-2013-3893, and CVE-2013-1690. Such exploits abuse in general the five steps

| Compiler | Passing *this* to virtual function via: | |
|---|---|---|
| | non-variadic function | variadic function |
| GCC (MinGW) | ecx | stack(FPO) |
| Clang (LLVM) | stack(FPO) | stack(FPO) |
| MS Visual C++ | ecx | stack(push) |

**Table 1:** Variations of step (4) based on compilers: variadic virtual functions retrieve the *this* pointer via the stack, either by a push instruction or by stack pointer addressing (FPO). For non-variadic functions, the `ecx` register is used.

outlined in Figure 1, but the actual manifestation of the steps can be completely different due to compiler optimizations and other low-level characteristics.

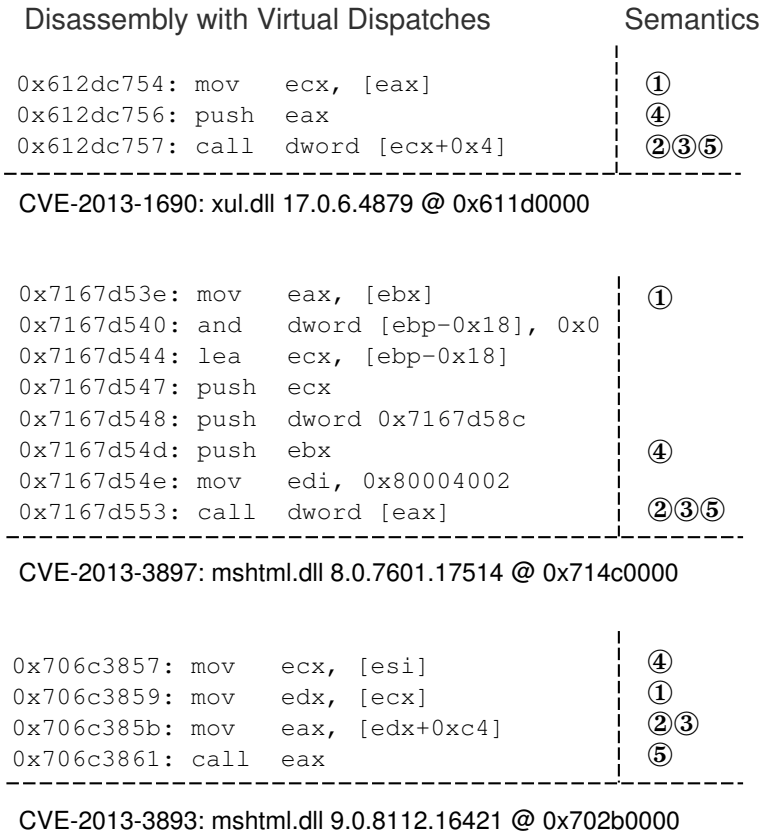


**Figure 2:** Disassembly and corresponding semantic steps of virtual dispatches in vulnerable modules with base addresses denoted after the @ sign. All examples were used in real attacks to gain control of the instruction pointer at step (5) by loading a fake vtable at step (1) first.

## 2.3 Threat Model: Vtable Hijacking

In the recent past, attackers have developed several exploitation techniques to turn use-after-free memory corruption vulnerabilities into reliable and arbitrary execution of their code of choice [46, 47, 50]. Such exploits render all current operating system security mechanisms ineffective and are one of the most common attack vectors we observe currently in the wild. In the following, we explain the different basic stages of such exploits based on Figure 3.

Use-after-free memory corruption vulnerabilities are based on dangling pointers. During runtime, a C++ application requests a new instance of class `C` at time $t_n$ which has implemented a

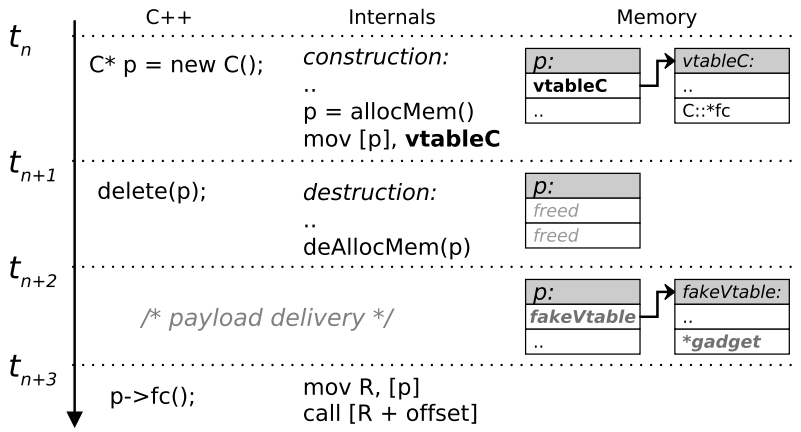


**Figure 3:** The C++ stages, internal low-level operations, and resulting instance's memory layout of a use-after-free exploitation process utilizing vtable hijacking

virtual function `fc`. Internally, constructing an instance invokes the memory manager to allocate needed memory. The instance's structure is built, including a vtable of `C`. Furthermore, a pointer `p` to the instance is created. At subsequent execution time $t_{n+1}$, the instance is removed but the pointer is still kept. If the programmer did not reset the pointer or if an alias was created, the reference to the freed memory still exists. Hence `p` or the alias becomes dangling.

At time $t_{n+2}$, an adversary can deliver payload content of her choice to the memory where `p` is pointing to (e.g., via heapspraying [17, 51] and similar techniques), and inject a fake vtable. In practice, this vtable resides in writable memory, whereas a legitimate vtable always resides in non-writable memory. Surprisingly, just checking for *non-writable* on vtable addresses during runtime before their usage already prevents many of the in-the-wild exploits as our evaluation shows (see Section5.3 for more details).

The value of the injected vtable needs to be carefully chosen by the attacker to have an entry pointing to the adversary's first chosen chunk of code [16]. Later at $t_{n+3}$, the virtual function `fc` is dispatched, leading to the instances's pointer `p` dereference, the fake vtable's pointer dereference, and the call of the adversary's code. Thus, this initiates the first step of a code execution attack by retrieving control of the instruction pointer and redirecting the logical program flow.

Note that an adversary can utilize code reuse methods to bypass *no-execute* (NX) protection and may craft memory leaks [47] beforehand to bypass *address space layout randomization* (ASLR). For more details, the reader is referred to the available literature on code reuse techniques [10, 15, 29, 43, 48]. In case *return-oriented programming* is utilized, the adversarie's first chosen code chunk is typically a so called *stack pivot gadget* used to exchange the stack pointer with a controlled register to further redirect program flow to her injected payload [16]. In real-world use-after-free web browser exploits, program snippets executed at time $t_n$, $t_{n+1}$ and $t_{n+3}$ often reside far away from each other and may have been generated from different source files. Also, mostly the pointer `p` is not the original created one but another reference pointer, which is reused.

Many recently detected zero day exploits utilize vtable hijacking to exploit vulnerabilities in web browsers as shown in Table 2.

| CVE | Targeted Application | Module | Vulnerability |
|---|---|---|---|
| 2013-1690 | Fx 17.0.6 (TorBrowser) | xul.dll | use-after-free |
| 2013-3893 | Internet Explorer 9 | mshtml.dll | use-after-free |
| 2013-3897 | Internet Explorer 8 | mshtml.dll | use-after-free |
| 2014-0322 | Internet Explorer 10 | mshtml.dll | use-after-free |
| 2014-1776 | Internet Explorer 8-11 | mshtml.dll | use-after-free |

**Table 2:** Zero-day attacks using vtable hijacking in-the-wild.

The main idea behind our protection scheme is to mitigate an exploitation attempt at the entry point, meaning, after the loading of a fake vtable pointer, but *before* it is used further to select a virtual function. Thus, the execution of the subsequent virtual call can be stopped, preventing the attacker from obtaining control of the instruction pointer, and impeding successive malicious computations.

### 2.4 Intermediate Language Prerequisites

As discussed in Section 2.2, dispatching a virtual function consists of several low-level instructions that can be interleaved by other code or distorted due to compiler optimization. Our goal is to identify such virtual dispatch sites in a given binary executable in an automated way. Since our target architecture is Intel x86, this is a complex task due to the large number of ways to express virtual dispatches in x86 assembly. Furthermore, side-effects of the individual instructions complicate the analysis process. Thus we opted to abstract away from the assembler level and perform our analysis based on an *intermediate language* (IL). In the following, we explain the needed background information and review the IL used for our implementation.

We utilize a RISC-like assembly language as IL to transform 32-bit x86 disassembly to an intermediate representation. Currently, our IL of choice is REIL [19]. As typical for RISC, there is only one dedicated memory load and memory write instruction. Thus, one x86 assembly instruction is typically transformed into several IL instructions. One IL instruction consists of a mnemonic and three operands. The first and second operand after the mnemonic represent the *source* and the mnemonic's preceding operand represents the *destination* holding the instruction's result value. Note that not all operands have to be used in one instruction. As registers, real x86 registers as well as an unlimited number of temporary registers (referred to as *IL registers*) can be used in an interchangeable way. Real registers are generalized to $R_i$ and temporary registers to $r_j$. We refer to an undetermined register (i.e., a register that is either $R$ or $r$) as $q$. Any immediate value is denoted with $m$ and operands which are either $q$ or $m$ are denoted with $v$.

Relevant instructions for our analysis are:

- memory load instruction $q_1 \leftarrow \mathit{load}\ v_1$ which loads a memory value pointed to by $v_1$ into $q_1$
- addition $q_1 \leftarrow \mathit{add}\ v_1, v_2$ which adds $v_1$ to $v_2$ and saves it to $q_1$
- substraction $q_1 \leftarrow \mathit{sub}\ v_1, v_2$ which substracts $v_1$ from $v_2$ and saves it to $q_1$
- register store $q_1 \leftarrow \mathit{stor}\ v_1$ which stores the value of $v_1$ into $q_1$
- memory store $v_2 \leftarrow \mathit{stom}\ v_1$ which stores the value $v_1$ to the memory pointed to by $v_2$
- *call* $v_1$ sets the instruction pointer to $v_1$.

Note that the usage of IL registers indicates that a complex x86 instruction was decomposed into several IL instructions. Decomposing an indirect addressing instruction with base and displacement will lead to several IL instructions with temporary registers. However, the semantic of a x86 indirect memory addressing can be achieved with several x86 instructions, too. When decomposing them to an IL, almost the same IL instructions are generated as before, except that *less* temporary and more real registers are used. This means that we can imply certain syntax usage in x86 disassembly from its IL representation. This becomes important in Section 4.1.

## 3 High-Level Overview

Developing a practical vtable hijacking mitigation and protection framework for binary C++ code involves several engineering challenges. In the following, we introduce our approach called T-VIP (*towards Vtable Integrity Protection*) to achieve this goal. T-VIP consists of vExtractor, the unit which identifies virtual dispatches in binary code, and PeBouncer which transforms the

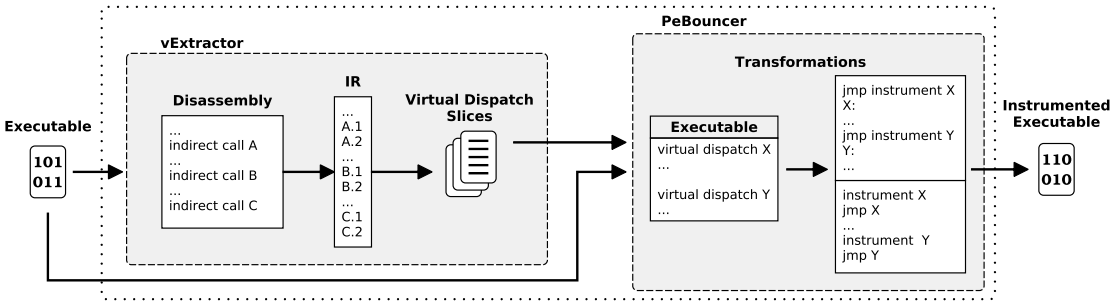


**Figure 4:** Overview of T-VIP, consisting of vExtractor and PeBouncer

original executable in order to allow only legitimate virtual calls to be executed. We start by giving a brief description of each component and present a high-level overview of their necessary interactions (see also Figure 4 for an illustration).

## 3.1 Automated Extraction of Virtual Function Dispatches

The first component of T-VIP is vExtractor, a static instruction slicing and extraction framework. It takes an executable as input whose vtable usage instructions before virtual calls should be protected, and disassembles it in a first step. While disassembling x86 binaries correctly is challenging [45], current approaches are sufficient to generate disassemblies usable for program transformations [59]. A control flow graph (CFG) is generated, reflecting the control flow in the form of vertices and edges. Vertices represent basic blocks and edges the control transfers between them. The disassembly is then transformed into an intermediate language to boil down the complex instruction set into a RISC-like syntax, while preserving the semantics and the CFG of the original code. Next, all addresses of indirect call instructions are extracted and defined as slicing criterions [9, 49]. vExtractor then performs backward program slicing on the IL to determine if an indirect call is a *virtual call*. It extracts all instructions which fulfill the low-level semantics of a virtual dispatch, thus, we retrieve virtual dispatch slices.

This is achieved via state machines, whereby one state consists of a set of IL instructions (see Section 4.1 for details). Furthermore, each state points to at least one successor state. Slicing starts at indirect call sites and the state whose instruction should be found next (in the backward IL instruction stream) is set as target state. If a target state is successfully matched against an instruction in the stream, then its successor is set as target state. As long as the last state of the state machine is not reached or is still satisfiable, slicing continues. It stops either if the last state is reached or if a state cannot be fulfilled. The latter disqualifies the indirect call site as a virtual call. In the former case, vExtractor classifies the call site as part of a virtual dispatch, extracts all instructions which are part of it, and associates its components such as x86 registers, offsets and addresses with instance, vtable and virtual function properties. Most important is the instruction which loads the vtable pointer of the instance into a register, as verification checks will be performed on these registers later on during runtime.

## 3.2 Automated Protection of Virtual Function Dispatches

The information produced by vExtractor and the original executable are the input to PeBouncer, the second component of T-VIP. We developed PeBouncer as a generic and static binary rewriting engine for executables conforming the PE specification [34]. Thus, while we use it to generate a protected executable, it is suitable to instrument instructions of interest similar to Pin [31] or DynamoRIO [5, 12]. Furthermore, it can be used to insert arbitrary code in order to enhance an executable with defense techniques similar to Vulcan [52] or SecondWrite [40].

A user who wishes to instrument an executable with PeBouncer has to specify the addresses of instructions to instrument. Each of these instructions is replaced by a forward jump redirecting

to an instrumentation stub inserted into a newly created section in the executable. The original instruction is preserved by copying it to the beginning of the stub. The stub ends with a backward jump targeting the address after the redirection to continue the original program flow. When replacing instructions, edge cases such as control transfers, basic block transitions, and relocations are considered and measurements are taken into account to rewrite such cases correctly (see Section 4.2).

The instrumentation can then implement an integrity policy $\mathscr{P}$ to perform checks on the virtual dispatches. Instrumentation stubs to enforce the policy can be developed in assembly code and can invoke functions of a user generated shared library, which we refer to as *service library*. Hence, major instrumentation code can reside inside the service library in order to remain customizable and still being able to accomplish complex tasks.

Using PeBouncer, we can instrument instructions which load a vtable in virtual function dispatches and generate distinct binaries with different integrity policies. As noted above, a vtable always resides in memory pages which are non-writable such as code or read-only data sections. They contain virtual function pointers pointing to read-only and executable pages. This basic insight can be leveraged to implement a simple integrity policy that checks if vtable pointers correctly point to non-writable memory pages. The following kinds of integrity policies are possible:

1. $\mathscr{P}_{nw}$: Look up vtable pointers in a lookup table with bits set for non-writable memory pages of modules and unset otherwise. This offers performant validation. When determining the memory protection of a vtable address, the page it belongs to is queried instead of the vtable pointer itself.
2. $\mathscr{P}_{nwa}$: Includes $\mathscr{P}_{nw}$. Additionally, one entry inside the vtable residing *above* the virtual function pointer about to be called, is randomly chosen. The entry is dereferenced, and the resulting address is queried for the non-writable flag. This is applied to all virtual dispatches calling a virtual function pointer that is not the first in the vtable.
3. $\mathscr{P}_{obj}$: Leverage type reconstruction of object-oriented code [26] in order to reconstruct all objects and the according class hierarchy. As an integrity check, we could verify if a vtable actually maps at virtual dispatch sites.

Note that $\mathscr{P}_{obj}$ is hard to implement in practice on binary code (in contrast to compiler-level implementation), as object recovery has yet to be shown practicable for huge COTS software like web browsers [26]. Hence, we did not implement this policy as part of our work.

The automated extraction of virtual dispatches can also yield slices not being virtual dispatches. For example, when binary code originated from nested C structs (see Section 5.1). As such, vExtractor might output instructions which seemingly load a vtable, but in fact represent other kinds of code constructs. This problem can be addressed with a profiling phase as follows: T-VIP first generates an executable instrumented with the checks using policy $\mathscr{P}_{nw}$, and runs it dynamically on tests in a trusted environment to visit (ideally) all instrumentation stubs. Hence, assumed vtables appearing in writable memory are the result of other constructs and are discarded in a second pass: To ensure vtable integrity protection at runtime, T-VIP applies PeBouncer a second time to the original executable using policies $\mathscr{P}_{nw}$ or $\mathscr{P}_{nwa}$ to produce the final protection.

Note that additional policy checks (even complex ones such as $\mathscr{P}_{obj}$) could be implemented in the future to ensure a more complete protection towards virtual table integrity, as PeBouncer is generic.

# 4 Implementation

We now describe in detail the inner workings of T-VIP involving the stages of disassembling an executable, transforming it into intermediate language, and performing program slicing to retrieve virtual dispatch slices. This is followed by the architecture of our generic binary rewriting engine, and its usage relevant to instrumenting and protecting executables against vtable hijacking.

Currently vExtractor supports the IDA Pro disassembler internally to disassemble an executable and generate a control flow graph. Other disassembly frameworks (e.g., BAP [13] and

ROSE [42]) could be supported in the future. Based on the disassembly, we search for call instructions with registers or indirect addressing with base register and optionally index, scale and diplacement as operand. Addresses of indirect calls are stored and the disassembly is transformed into the platform-independent intermediate language REIL [19]. This produces a second CFG layer: Even strict x86 basic blocks having only one entry and exit can turn into a CFG when being transformed into an IL. Due to certain x86 instructions having implicit branches, the decomposed IL instructions emerge into a CFG, such that a x86 basic block is represented as CFG. We treat the outer x86 CFG layer and inner IL CFG layer separately.

As discussed in Section 2.2, virtual dispatches have a certain semantic which can express itself differently in the x86 syntax. We exploit the advantages of an IL which converts syntactically greatly varying but semantically similar instruction streams into similar constructs. This facilitates the harvesting and classification of semantically similar but syntactically different instruction streams like virtual dispatches via backward slicing.

## 4.1 Amorphous Slicing

We implemented intra-procedural backward slicing on IL into vExtractor based on state machines. As program slicing is a common technique, we refer the reader to the literature for information about program slicing [9, 23, 49].

In our state machines, one state consists of a set of IL instruction patterns. Figure 5 shows our state design and a state transition. When an IL instruction in the instruction stream to search (①), fulfills a pattern's condition such as matching mnemonic and matching destination register, the state is triggered (②). The source of the matched instruction is taken and inserted into the successor state's instruction patterns as destination (③). A transition to the successor state is performed (④). Slicing continues (⑤), and the patterns of the successor state have to be fulfilled in order to trigger it.

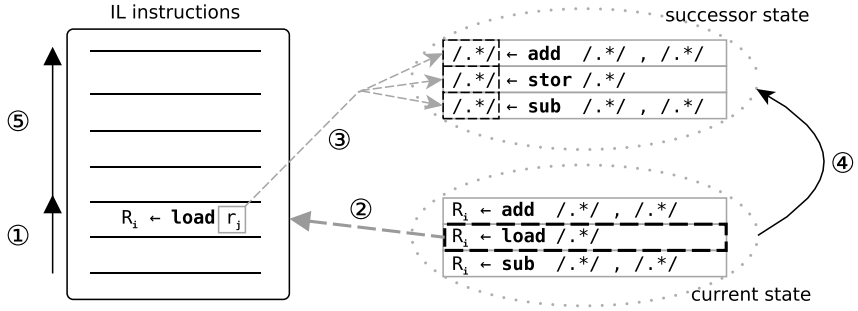


**Figure 5:** State design and state transition principle. Wildcard operands are denoted with `/.*/`

While the semantics of virtual function dispatches are simple, they can manifest themselves in different and complex x86 syntax constructs (as previously shown in Figure 2). To extract virtual dispatches precisely, we developed several state machines to unveil instructions being components of them, each visualized in Figure 6.

The first state machine is used to search backwards, starting from each indirect call instruction in the IL instruction stream for dereferences. On a successful pass, if the final state is reached, vExtractor detects the vtable entry's dereferencing and the instance's dereferencing. We refer to the instance also as *obj*. Furthermore, if available in the IL instruction stream, the reference to the instance and the reference to the reference are detected, too. We name the reference which—when dereferenced—yields a reference to the instance, `ref2`. The reference that yields the address of the instance when dereferenced is called `ref1`. The instance's reference (`ref1`), and also the reference's reference (`ref2`), become important in subsequent state machines.

In the first state machine, the indirect call represents the start state. State one to match is a memory load: the call's destination register is followed to its definition. In case it was defined by a memory load, state two is set as next state to match. The matched instruction's source of state

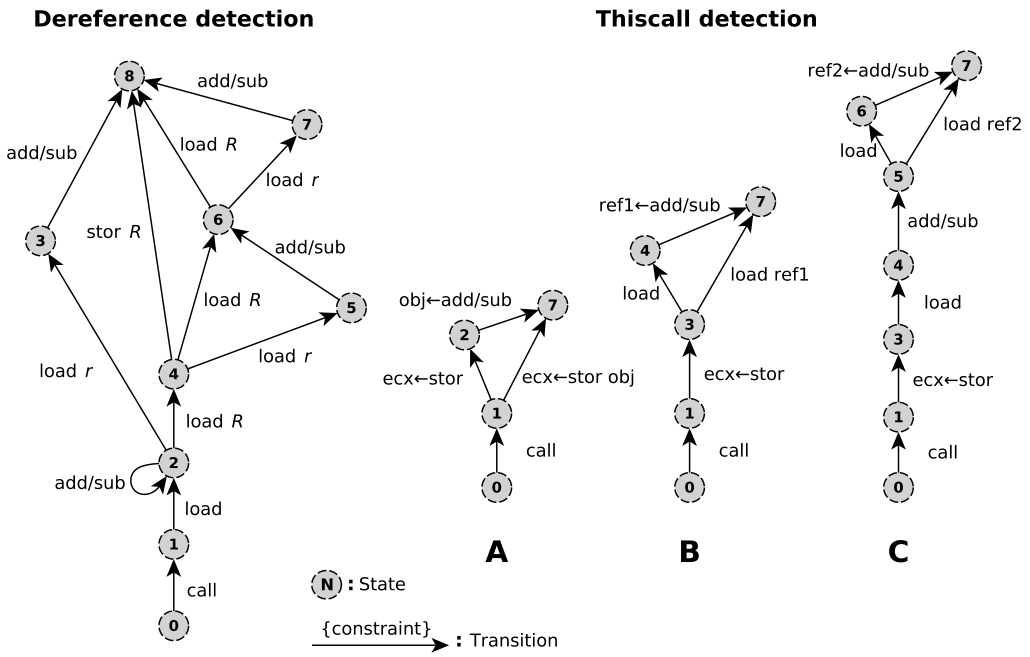


**Figure 6:** State machines used to detect consecutive dereferences and *thiscall* in virtual dispatches. Transitions may be constraint (by e.g.: mnemonic: *add*, source: *add v*, destination: $v \leftarrow add$)

one becomes the destination register of state two. A mandatory memory load and an optional addition or subtraction instruction are searched. In case an addition or subtraction is matched, the state to match next does not change. As soon as the memory load is matched, the source register in the matched instruction dictates the next state to match: In case it is an IL register, state three is set as next state. If it is a x86 register, state four is the next state to match. To trigger state eight from state three, the destination register of state three has to be found as source register in an addition or subtraction instruction respectively. State eight is the final state. Thus, starting from state zero (start state), successively transitioning into states 1, 2, 3 and ending at state eight yields the following IL example slice:

| State: | | | | | |
|---|---|---|---|---|---|
| 3 | $r_0$ | $\leftarrow$ | **add** | $q_3$, | $m_1$ |
| 2 | $q_2$ | $\leftarrow$ | **load** | $r_0$ | |
| 2 | $q_1$ | $\leftarrow$ | **add** | $q_2$, | $m_0$ |
| 1 | $q_0$ | $\leftarrow$ | **load** | $q_1$ | |
| 0 | | | **call** | $q_0$ | |

When extracting the x86 disassembly, corresponding to the IL slice, a dereferencing sequence arises, which exists in virtual dispatches when used with multiple inheritance:

**mov** $q_2, [q_3 + m_1]$
**call** $[q_2 + m_0]$

$q_3$ is the instance address ($obj$), $q_2$ contains the vtable address, $m_1$ is the displacement to the base classes' vtable to take, and $m_0$ is the offset to the virtual function to call.

When vExtractor backtraces the IL instruction stream continuing from state two and transitions the other states until the final state, it detects the following additional information:

| Transitions | Information |
|---|---|
| $2 \rightarrow 4$ | register with `obj` |
| $4 \rightarrow 8$ | second register with `obj` |
| $4 \rightarrow 6$ | register with `ref1` |
| $4 \rightarrow 5 \rightarrow 6$ | register and displacement of `ref1` |
| $6 \rightarrow 8$ | register with `ref2` |
| $6 \rightarrow 7 \rightarrow 8$ | register and displacement of `ref2` |

While the first state machine extracts semantic components related to instance and vtable loading, the second state machine is designed to detect a *thiscall*. In a *thiscall*, the (adjusted) instance's address is passed via `ecx` to the virtual function as a hidden parameter. Dependent on the information gained by slicing with state machine one, the second state machine is built dynamically and adjusted with the registers and offsets found by machine one. Thus, one of three state machines shown in Figure 6 arises and will be chosen for the *thiscall* detection.

Machine **A** is chosen if `ref1` and `ref2` were not found. Thus, it detects if `obj` is moved into `ecx` (transitions $0 \rightarrow 1 \rightarrow 7$). With transitions $0 \rightarrow 1 \rightarrow 2 \rightarrow 7$, **A** detects if `ecx` is filled with an adjusted instance's address. That is the case in virtual dispatches used in multiple inheritance, where an adjusted instance's address is dereferenced to gain the vtable, and as well moved to `ecx` to prepare the *thiscall*.

Machine **B** is chosen if there is no `ref2`, and it detects if `ref1` is dereferenced to `obj` and if a subsequent move into `ecx` follows (transitions $0 \rightarrow 1 \rightarrow 3 \rightarrow 7$). If `ref1` was displaced by an offset, transitions $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 7$ of **B** detect the *thiscall*. If it fails to detect a thiscall with `ref1`, but recognizes that `obj` was used in transition $0 \rightarrow 1 \rightarrow 3$, it is declared also as valid *thiscall*.

When `ref2` was found, machine **C** is chosen. Similar to **B**, it detects if a *thiscall* is prepared using `ref2`. If it fails to reach the final state but recognizes that `obj` or `ref1` was used for the *thiscall*, it is classified as valid.

Additional state machines are used to detect if an instance is passed as first parameter via the stack to the virtual function at dispatch time. They comprise our third set of state machines. These are built similar to **A**, **B** and **C**. But instead of using states with `ecx` $\leftarrow$ *stor* $q_i$, states containing patterns with `esp` $\leftarrow$ *stom* $q_i$ are utilized. Thus, if the first parameter on the stack is `obj`, or if it evolves from `ref1` or `ref2`, then the corresponding instructions are classified as components of a virtual function dispatch.

**Summary** vExtractor uses a state machine based approach to harvest virtual dispatch data of potential virtual dispatches. Therefore, it walks the CFG of a binary of interest backwards, starting from indirect calls and tries to match states to IL instructions. On a match, object location and register, vtable register, virtual function offset, and the addresses of corresponding instructions are saved. Additionally the corresponding disassemblies of virtual dispatches are gained as slices. This virtual dispatch data is then fed into PeBouncer to generate an instrumentation of the vtable loading instruction to enforce specific policies.

## 4.2 Binary Transformations

*Static binary rewriting* (also called *static binary transformations*) allows the modification of compiled executables without the need for source code information or recompilation, and is done directly on the binary level. We implemented PeBouncer as a generic and automated instrumentation engine for PE executables for Windows. Thus, transformations are applied statically to produce an instrumented binary that realizes an integrity policy. The relevant enforcement checks become active during runtime.

### 4.2.1 Insertion of Instrumentation Checks

We statically create a new code section where the integrity policy checking code will reside. Addresses of instructions to instrument are fed into PeBouncer and disassembled. As an instruction will be replaced with a redirecting, relative 32-bit jump to its instrumentation stub, we need

enough space for the jump opcodes, while keeping other instructions in the neighborhood functional. Therefore, if the size of the original instruction's opcodes is greater than the jump, we replace it and insert NOPs to fill the remaining space. If the size is smaller than the size of the jump, we disassemble downwards starting from the original instruction until there is at least enough space to insert our jump. Thus, we replace the instruction to instrument and subsequent instructions. The replaced instruction(s) will be copied to the beginning of the associated instrumentation stub into the new section.

Note that some instructions cannot be overwritten without additional measures:

1. Targets of all *relative control transfer* instructions have to be kept
2. Instructions with *relocations* must remain relocatable, our jump should not change due to an original relocation
3. Basic block terminators and basic block leaders must be preserved. For example, one terminator's opcodes, and at the adjacent address located leader opcodes cannot be replaced with opcodes of one instruction. When there are several entries into the adjacent block, the inserted instruction would get split and resulting opcodes of both halves get interpreted wrongly as distinct instructions.

To tackle these and similar corner cases, we stop our downward search for space at such instructions and traverse the disassembly upwards instead, starting from the instruction to instrument. Similar to the downward search, we stop as soon as we have enough space to insert our redirecting jump, while overwriting additional instructions if necessary. If the instruction to instrument is *enclosed* between two instructions of above mentioned edge cases, we overwrite one of them with an illegal instruction and install a vectored exception handler. It then serves as a trampoline to the instrumentation stub.

For each instruction to instrument, the redirecting jump's target address is automatically calculated to point to the next available free location in the new section. The replaced instructions are copied there, and the instrumentation code is placed below them into the stub. At the end of the stub, a relative 32-bit jump is inserted. Its target address is calculated to point to the original code, to the address right after the redirecting jump. Thus, the new code section is filled successively, stub after stub. An inserted instrumentation stub is shown in Figure 7.

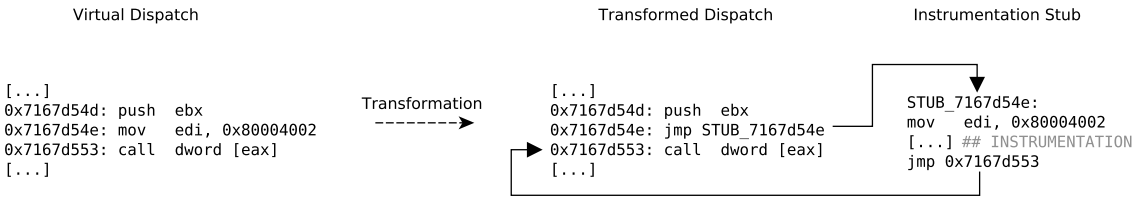


**Figure 7:** Instrumentation stub insertion: a virtual dispatch is transformed in order to perform instrumentation on the vtable register (`EAX`) before a virtual call is issued.

vExtractor provides virtual dispatch slices including the addresses of vtable load instructions. For a C++ executable, we utilize these addresses to instrument and to protect them.

### 4.2.2 Generation of Instrumentation Stubs

For instructions of interest, instrumentation stubs can be supplied in position-independent assembly. Hence, relative addressing can be utilized. A stub starts with a prolog to save the register context and ends with an epilogue to restore it. The Netwide Assembler (NASM [55]) is used as assembly backend. We created an annotation feature that serves PeBouncer as hint to modify the stub *after* it is assembled, but *before* it is inserted into our new code section. This allows one-time assembling of instrumentation code and many-time stubwise modification.

It works as follows: Instrumentation code is provided as assembly file and contains specific keywords inside angle brackets, which PeBouncer recognizes. The brackets including the annotation keywords are replaced with a x86 mnemonic and a hash of the keyword as operand. Depending on

the keyword, corresponding mnemonics which allow at least a four byte operand are used. This way, the assembly syntax stays error free, and the keyword's information is preserved. Also, the occurrence of each keyword is counted. After the instrumentation code is assembled, it contains the binary representation of the hashes. Before the binary instrumentation code is about to be inserted as a stub, the hashes are searched and their occurrences are compared to the keywords' occurrences to prevent collisions. Then they are replaced with adjusted opcodes specific for a keyword and specific for an instrumentation stub. The reader may ask what benefit it has. We instrument instructions representing vtable loading and each of these may use a different mnemonic and a different register to load the vtable. With our annotation feature, PeBouncer can assemble one instrumentation code for all instructions to instrument, and modify it for each stub to include the specific register which holds the vtable. Thus, each instrumentation stub for every vtable load instruction will operate on its specific vtable register.

While creating instrumentation code in assembly is already convenient, the complete API provided by an operating system can be used as well. This is important for the usage of Windows API functions (e.g., `OutputDebugString`). Therefore, a shared library (*service library*) is compiled with exported functions which wrap the API functions to use. Then an instrumentation stub is developed in a certain structure. It starts with a prolog to save the register context and instructions follow which save the instruction pointer (IP) to the stack. Thus, the first stack value will point to the absolute virtual address of the beginning of the stub. To call a service library's function from instrumentation code, an instruction is specified to load the saved IP into a register. A subsequent indirect call with the register and an annotation keyword containing the library and function name follows. When PeBouncer encounters such a keyword, it replaces the keyword with its four byte hash, such that an indirect call instruction with base register and displacement emerges. The structure of a possible instrumentation stub is shown in Figure 8.

```
#### INSTRUMENTATION START ####
## Prolog
pushad
pushfd
call $+5
sub dword [esp], 7 ## calculate stub IP
## --------

## annotation: move register with vtable to ecx
<MOV_ECX_VTABLE_REG>

mov ebp, dword [esp]  ## get IP

push ecx ## push vtable address

## annotation: service lib function (outputs vtable)
call dword [ebp + <IAT.vProtSrv.debugOut>]

add esp, 4 ## remove vtable address

## Epilogue
add esp, 4
popfd
popad
## --------
#### INSTRUMENTATION END ####
```

**Figure 8:** Instrumentation stub structure: During runtime, the context is saved in the prolog and restored in the epilogue. Annotation keywords are replaced with respective instructions before assembling and stubwise modified before insertion.

After the instrumentation code is assembled, the four byte hash is replaced with a binary displacement value. This value is calculated in such a way that the register with the saved IP and the displacement (when summed up) point to a custom *Import Address Table* entry. This entry will contain the address to the service library's function to use. During runtime – similar to an Import Address Table (IAT) generation – an additional data section is filled with pointers to the service library's functions. This resolution is performed as soon as the instrumented executable is

loaded into the address space of an application. Thus, the instrumentation code in the executable can reference the service library's resolved functions addresses in our custom generated IAT. The service library itself can be loaded in many ways into an applications address space [24]. Either the application's entry point is patched to load the library in case the instrumented executable is known to be loaded *afterwards*, or the service library is specified to be loaded into every process' address space [32] in case the instrumented executable is known to be loaded at the application's startup. In any case, the service library is loaded *before* the instrumented executable's entrypoint is about to be executed. Note that the above concept allows the full support of both ASLR and DEP (NX).

#### 4.2.3 Virtual Dispatch Instrumentation

To mitigate vtable hijacking, virtual dispatches are instrumented with policies $\mathscr{P}_{nw}$ (12 assembly instructions) or $\mathscr{P}_{nwa}$ (23 assembly instructions). Each virtual dispatch consists of the low-level semantic steps described in Section 2.2. To protect against the use of fake vtables, we instrument vtable load instructions after step one of the virtual dispatch semantic to be able to check the register with the vtable address. We do this in the following way: We keep a read-only 64KB sized lookup bitmap in our service library, representing the complete usermode memory pages. This bitmap is made writable and set up when an instrumented module is loaded into the address space of its application. Then its access permissions are set to read-only again. Each bit represents the write permissions of a page. A set bit means the page is non-writable, an unset bit means the page is writable. Thus, when loading the module, we find all non-writable module sections in complete memory and set the appropriate bits of corresponding pages in the bitmap. To keep it up to date, Windows loader functions are hooked to change bits when unprotected modules are loaded and unloaded. By now, instrumentation checks can query the page of a vtable address by a simple lookup instead of querying the vtable itself: During runtime, a vtable is loaded into a register. The control flow is then rerouted to its instrumentation stub. Then, the vtable address is transformed with simple operations to an index into the page bitmap. The bit for the page is queried and if it is not set, a violation of $\mathscr{P}_{nw}$ occurred. A set bit means that the page, and thus the vtable it resides in, is non-writable. However, an adversary could circumvent this check, if she manages to find an address which is non-writable and contains a pointer to a gadget of choice to start her ROP chain. Thus, to mitigate this type of attacks, after the page lookup of the vtable, there is an additional virtual method check ($\mathscr{P}_{nwa}$): As step two of virtual dispatch semantics provides the offset to the virtual function, a pseudo-random index up to that offset is generated with the help of `rdtsc`. The vtable is dereferenced at this index and the resulting value is looked up in the page bitmap. A violation can be detected, as *all* entries in a valid vtable *above* the offset of the virtual function about to be called are method pointers pointing into non-writable code pages.

# 5 Evaluation

We have implemented prototypes for both vExtractor and PeBouncer. In what follows, we evaluate both tools regarding their precision, performance overhead, and prevention of real-world exploits.

## 5.1 vExtractor's Precision

As a first step, we wanted to gain insights into the precision and recall of vExtractor's virtual dispatch detection. The analysis is performed against all identified indirect call instructions of a given program. We leverage a simple classification metric which states that any virtual dispatch found not being a virtual dispatch is a false positive ($FP$) and missed virtual dispatches are false negatives ($FN$). True positives ($TP$) and true negatives ($TN$) are the correctly found and rejected virtual dispatches, respectively. Based on this, we can define *precision*, *recall*, and *F-measure* as follows:

$$\begin{aligned} \text{precision} &= \frac{TP}{TP+FP} \\ \text{recall} &= \frac{TP}{TP+FN} \\ \text{F-Measure} &= 2 \cdot \frac{precision \cdot recall}{precision + recall} \end{aligned} \tag{3}$$

We used version 4.9.0 of the *MinGW-w64* GCC cross compiler as it contains a partly implementation of GCC's *virtual table verification* feature [56]. Also, we ported missing parts of GCC's vtable verification library (`vtv`) to Windows to be able to compile 32-bit PE files with MinGW-w64 resulting in functional vtable verification checks. This porting was necessary since we instrument programs as proprietary web browsers such as Microsoft's Internet Explorer on Windows (see Sections 5.2 and 5.3). Thus, we needed a `vtv` version on Windows to compare against[1].

Compilation of 32-bit PE files with the `-fvtable-verify` flag will insert verification calls at each virtual dispatch after the instruction which loads the vtable into a register. Other resulting code stays identical for the same program when this flag is omitted. The version with verification is used to build a ground truth for indirect calls as they are preceded with verification routines in case the indirect call is part of a virtual dispatch. Otherwise, they are non-virtual calls. Indirect calls in the version compiled without verification are then grouped exactly into virtual dispatches and non-virtual dispatches based on the information gained from the first version. We applied vExtractor to the second version and classified the outcome of slices function-wise to retrieve a classification on the binary.

We utilized the open source C++ cryptographic library Botan [11], which contains 90 cryptographic primitives. We chose Botan because of its extended use of C++ features. We compiled it with and without vtable verification. vExtractor traced a total of 6779 indirect calls and identified 6484 virtual dispatches ($TP$) with 62 being non-virtual dispatches and 179 false negatives. This yields a precision of 0.99, a recall of 0.97, and an F-measure of 0.98.

We analyzed the reasons for false positives and discovered that they are due to C code constructs. More specifically, C code can have semantics equal to virtual dispatches. Consider the following C code line with `st` and `innerSt` being pointers to structs and `sFn` being a function pointer:

```
st->innerSt->sFn(st, p1, p2)
```

The two dereferences and `st` as first parameter fulfill the virtual dispatch semantics when compiled down to binary code. However, these and similar constructs can be eliminated in a profiling phase, as we show later in Section 5.2. We discuss the reasons for false negatives in Section 7.

## 5.2 Runtime of Instrumented Programs

To assess the performance overhead, we compiled Botan with and without GCC's vtable verification and compared the runtime in micro- and macro-benchmarks to the plain build of Botan, and to the plain build instrumented with T-VIP. Additionally, we compiled the SPEC CPU2006 benchmark with MS Visual C++, instrumented it with T-VIP and compared the runtimes to the native build. Finally, we hardened browser modules and measured their runtime overhead. Benchmarks were performed on an Intel Quad Core i7 at 2.6GHz with 2 GB of RAM running Windows 7.

**Comparison to GCC's virtual table verification** We patched our port of `vtv`'s source code to measure the CPU cycles needed for each verification routine execution (`VLTVerifyVtablePointer`) in order to perform micro-benchmarks. Therefore, we inserted GCC's build-in `rdtsc` routine at

[1] Our MinGW-w64 extension is available at `https://github.com/RUB-SysSec/WindowsVTV`

the beginning and at the end of the verification routine and executed Botan's benchmark. The verification produced a median cycle count of 9205. We binary-rewrote Botan using T-VIP to measure the cycle count of our vtable protecting check code. Thus, we added additional `rdtsc` calls to the start and end of our instrumentation checks consisting of policies $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$, and took the vanilla build of Botan. We ran the benchmark and retrieved a median cycle count of 8,225 for $\mathscr{P}_{nw}$ and 12,335 for $\mathscr{P}_{nwa}$.

To perform macro-benchmarks, we built Botan with and without `vtv` with our newly ported GCC, not using `rdtsc`. We protected the vanilla build with T-VIP using policy $\mathscr{P}_{nwa}$ and run the benchmarking capability of both, ten times each. Botan's benchmark consists of 90 highly demanding cryptographic algorithms. The version compiled with GCC `vtv` produced a median overhead of1.0 % with 46 algorithms producing a median overhead smaller than 2.0 %. The version protected with T-VIP produced a median overhead of 15.9 % with 37 algorithms producing a median overhead smaller than 2.0 %. We investigated the rather high appearing overhead: T-VIP installs a vectored exception handler for instrumented instructions, which cannot be overwritten with a jump to an instrumentation stub (see Section 4.2.1). As an exception handler produces high overhead, algorithms executing it will run perceptibly slower.

**Runtime overhead measurements** We compiled the seven C++ benchmarks of SPEC CPU 2006 with MS Visual C++ 2010, applied vExtractor and gained virtual dispatch slices for all except two (i.e., only five benchmarks actually have virtual dispatches). We hardened them with policies $\mathscr{P}_{nw}$, $\mathscr{P}_{nwa}$, and an empty policy ($\mathscr{P}_{e}$) separately, using PeBouncer. $\mathscr{P}_{e}$ consist of a prolog and epilog only, and serves to measure the net overhead introduced by PeBouncer. The results are depicted in Table 3.

| CPU2006 | Size | #VD | Runtime (in s) and overhead (in %) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Native | $\mathscr{P}_{e}$ | | $\mathscr{P}_{nw}$ | | $\mathscr{P}_{nwa}$ | |
| | | | rt(s) | rt(s) | ov(%) | rt(s) | ov(%) | rt(s) | ov(%) |
| soplex | 403K | 746 | 232.25 | 231.05 | -0.52 | 232.41 | 0.07 | 233.64 | 0.60 |
| omnetpp | 793K | 1593 | 217.12 | 293.72 | 35.28 | 303.48 | 39.78 | 318.15 | 46.53 |
| povray | 1038K | 154 | 164.27 | 164.22 | -0.03 | 164.36 | 0.06 | 164.31 | 0.03 |
| dealII | 947K | 272 | 360.97 | 361.75 | 0.22 | 363.01 | 0.57 | 363.14 | 0.60 |
| xalancbmk | 3673K | 14061 | 182.97 | 294.29 | 60.84 | 331.98 | 81.44 | 372.26 | 103.45 |

**Table 3:** Binary sizes, amount of instrumented virtual dispatches (#VD), median runtime over three runs, and overheads of C++ SPEC CPU2006 benchmarks.

Overheads are $\leq 0.6\,\%$ in `soplex`, `povray` and `dealII`, while high overheads for $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$ in `omnetpp` and `xalancbmk` are mostly due to the net overhead of our rewriting engine ($\mathscr{P}_{e}$ column in Table 3). Using our policies $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$ with another binary rewriter could lower the overhead. However, as we show with COTS browser modules, the overhead in macro-bechmarks is actually low in practice.

We applied vExtractor to `xul.dll` of Mozilla Firefox 17.0.6 and to `mshtml.dll` of Internet Explorer in versions 8, 9, and 10. We chose these modules because they contain the major parts of the browsers' engines and former zero-day attacks were related to code in these modules (see Section 5.3 for details). Table 4 shows the amount of indirect calls and extracted virtual dispatch slices.

We applied PeBouncer to each module to instrument all vtable load instructions, such that during runtime, the addresses of vtables, their memory page permission, and the addresses of the corresponding virtual call sites are gained with `OutputDebugString`. Less than 900 exception handlers had to be inserted for each module due to non-overwritable instructions, but *all* were instrumented without problems.

We then, at first, ran the two browser benchmarks *SunSpider* [4] and *Kraken* [38] to profile the browsers. Vtable addresses retrieved, not being vtables, show themselves as writable. This

way, we can filter out all non-virtual dispatches like calls from nested C structs and eliminate all false positives (see Table 4 for details).

| App. | Module | #IC | #Slices | #Filtered | #Instr. |
|---|---|---|---|---|---|
| Fx 17.0.6 | xul.dll | 66,120 | 53,268 | 73 | 53,195 |
| IE 8 | mshtml.dll | 23,682 | 19,721 | 3,117 | 16,604 |
| IE 9 | mshtml.dll | 64,721 | 53,312 | 7,735 | 45,577 |
| IE 10 | mshtml.dll | 56,149 | 44,383 | 5,515 | 38,868 |

**Table 4:** Amount of indirect calls (#IC), extracted virtual dispatch slices, and filtered non-virtual calls. #Instr. indicates the number of slices fed into PeBouncer to harden listed modules.

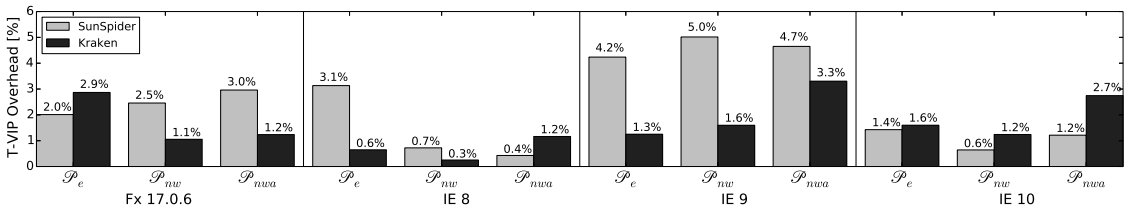


**Figure 9:** Runtime overhead for instrumented browsers on the browser benchmarks *SunSpider* and *Kraken*.

These were removed from the virtual dispatch slices and each module was rewritten again by PeBouncer. This time, we used instrumentation checks based on our polices $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$, and policy $\mathscr{P}_e$ was used as well. All benchmarks were run again to measure the introduced performance overhead. The results can be seen in Figure 9 and yield an overall average performance overhead of approx. 2.1% ($\mathscr{P}_e$), 1.6% ($\mathscr{P}_{nw}$) and 2.2% ($\mathscr{P}_{nwa}$).

### 5.3 Vtable Hijacking Detection

Real-world exploits for zero-day vulnerabilities utilized vtable hijacking to first load a fake vtable, and then dereference an entry to call a ROP gadget. In this way, attackers gained a foothold into victim systems via CVE-2013-3897, CVE-2013-3893, and CVE-2013-1690. The virtual dispatches were all found by vExtractor and successfully protected with policies $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$ by PeBouncer. We then attempted to exploit the protected web browsers with corresponding exploits from Metasploit and exploits gained from the wild. All attempts were detected successfully already with $\mathscr{P}_{nw}$.

Another critical vulnerability (CVE-2013-2556) in Windows 7 allowed remote code execution without any shellcode or ROP in conjuction with vtable hijacking. The culprit was the non-ASLR protected *SharedUserData* memory region containing function pointers [58]. Attackers used the region's address as fake vtable and an entry with a pointer to `LdrHotPatchRoutine` to gain remote code execution via virtual dispatches. This is detected by policy $\mathscr{P}_{nw}$, as it checks vtables for non-writable in modules. Another zero-day use-after-free vulnerability (CVE-2014-0322) was used in targeted attacks. While the vulnerability only allowed a one byte write, a vtable pointer of a *flash* object was modified to gain control [20]. As the precision of vExtractor is high, T-VIP can protect against this vulnerability when the corresponding virtual dispatch is extracted and then instrumented by PeBouncer.

## 6 Related Work

Due to their prevalence and high practical impact, software vulnerabilities have received a lot of attention in the last decades. Many different techniques were proposed to either exploit or

detect/mitigate/prevent them. In the following, we briefly review work that is closely related to our approach and discuss how our approach differs from previous work in this area.

**Reducing the Attack Surface** There are many methods that can be used to harden a given system against software vulnerabilities. A few examples include *data execution prevention* (DEP) [36], *address space layout randomization* (ASLR), SAFESEH/SEHOP to protect execption handlers, *instruction set randomization* (ISR) [6], and similar approaches [21, 28, 57]. They are all complementary to our approach, which primarily focusses on protecting the integrity of vtables.

*Reference counting* is a memory management technique used for example in garbage collectors to track during runtime how many references to a given object exist [30, 44]. The basic insight is that if no pointer to an object exists anymore, the object can be safely freed. Unfortunately, referencing counting induces a certain performance overhead and no security guarantees can be given since an attacker might be able to corrupt the reference counts since this information needs to be stored on the heap.

**Control Flow Integrity (CFI)** A general concept to prevent memory corruption attacks that divert the control flow of a given program is *Control Flow Integrity* (CFI) [1]. The basic idea is to instrument a given program to verify that each control flow transfer jumps to a valid program location. Recently, several methods were proposed to implement CFI with low performance overhead [59, 60]. Efficient implementation incur a performance overhead of less than 5%, but had to sacrifice some of the security guarantees given in the original CFI proposal [1]. Götkas et al. recently demonstrated circumventions of these CFI implementations [22]. Their proof-of-concept attack gains control over an indirect transfer by overwriting a vtable pointer with a heap address. This specific use case is detectable by our approach: we enforce policies at instructions which load vtable addresses *before* targets of indirect transfers are loaded. Bogus targets might seem legitimate in coarse-grained CFI protections, due to conforming to their CFI policies. We detect a violation if *any* indirect target comes from a fake vtable.

The main difference compared to existing work is that we specifically focus on the integrity of virtual dispatches, since vtable hijacking attacks have become one of the most widely used attack vectors recently. Instead of protecting all indirect jumps and inducing a performance impact that prevents widespread adoption [54], we focus on a specific subset of indirect jumps that are an attractive target for attackers.

**Compiler Extensions** Recently, several compiler extensions were proposed that protect vtables from hijacking attacks:

- GCC introduced the `-fvtable-verify` option [56] that analyzes the class hierarchy during the compilation phase to detect all vtables. Furthermore, checks are inserted at all virtual function call sites to verify the integrity of virtual method dispatches.
- Closely related, SafeDispatch implements an LLVM extension that performs the same basic steps [25]. A class hierarchy analysis is used to determine all valid method implementations and additional checks are inserted to ensure that only valid dispatches are performed during runtime. The measured runtime overhead is about 2.1%.
- VTGuard by Microsoft [27] adds a guard entry at the end of the vtable such that (certain kinds of) vtable hijacks can be detected. This instrumentation is added during the compilation phase. If an information leak exists, an attacker could use this to obtain information about the guard entry, enabling a bypass of the approach.

The main difference to our approach is the fact that we operate on the binary level such that we can also protect proprietary programs where no source code is available. Since the full class hierarchy can be determined during the compilation phase, the security guarantee provided by the first two approaches is stronger than ours: these approaches can implement $\mathscr{P}_{obj}$ and perform

a full integrity check. However, empirical results demonstrate that our policy can already defeat in-the-wild zero-day exploits. Our performance overhead is sligthly higher, but this is mainly due to the fact that we instrument binary programs.

**Heap Monitoring** By monitoring the heap of a given program during execution, dangling pointers that can lead to use-after-free or double-free vulnerabilities can be detected. For example, UNDANGLE is a detection tool that leverages taint and pointer tracking to recognize dangling pointers during runtime [14]. Note that the tool is not designed to protect applications. CLING is a memory allocator that constrains memory allocation to allow address space reuse only among objects of the same type, thus preventing use-after-free vulnerabilities [3]. The authors report a performance overhead of less than 2% for most benchmarks. Other proposals for memory allocators that provide additional security guarantees are DIEHARD [8] and DIEHARDER [39]. Both prevent use-after-free vulnerabilities, but have rather high performance overheads (e.g., DIEHARDER imposes on average a 20% performance penalty). Our performance overhead is comparable to CLING, but we do not require to exchange the memory allocator.

# 7 Discussion

In the following, we discuss the limitations and shortcomings of our approach and the current implementation.

It is crucial to identify virtual dispatches precisely in order to protect all virtual call sites. As the evaluation shows, VEXTRACTOR misses 2.6 % of virtual dispatches. Recall formulae (1) and (2) from §2.2. Manual investigation revealed that in rare cases, especially GCC creates multiple *aliases* for *obj*. While VEXTRACTOR already performs an alias analysis to some extent, cases can slip away if an alias of *obj* is used in instructions represented by (2), but cannot be connected to *obj* in (1). Also, at the time of writing, trying to compile Firefox with the original GCC 4.9.0 enabling GCC's vtable-verification, led to compiler crashes. Thus, we were not able to evaluate VEXTRACTOR's precision using Firefox as ground truth.

Currently, binaries have to be profiled in order to remove virtual dispatch-like code constructs. On the binary level, it is impossible to differentiate certain C code constructs from virtual dispatches, and thus we need this (automated) profiling phase, to filter all non-virtual dispatches.

As shown in our evaluation, T-VIP protects against real-world vtable hijacking attacks. However, policy $\mathscr{P}_{nw}$ could be circumvented by using a pointer residing in a non-writable module memory page and pointing to code of an attacker's choice. To mitigate this, we sacrify performance by generating a random index into a vtable in the implementation of $\mathscr{P}_{nwa}$. Hence, T-VIP guarantees that a *different* vtable entry is checked for *each* execution time at the *same* virtual dispatch. An attacker is thereby restricted to use non-writable function tables in order to reliably compromise a system. By itself, circumventing this is already very hard, but would be still possible if a valid vtable of a wrong class type is used at a virtual dispatch site. This is a limitation we have in common with VTGUARD according to [25]. However, implementing $\mathscr{P}_{obj}$ would prevent even such attacks.

PEBOUNCER currently supports 32-bit PE files since the majority of web browsers uses 32-bit code and this is the primary target of use-after-free exploits. However, the concept of PEBOUNCER is usable for 64-bit code and the ELF file format as well, with only minor modifications. Some corner cases during rewriting are currently handled by an exception handler and introduce additional overhead (see Section 4.2.1). This could be solved by leveraging binary rewriting capabilities of ROSE [42] to insert checks inline.

# 8 Conclusion

In this paper, we introduced an approach to protect binary programs against vtable hijacking vulnerabilities, which have become the de-facto attack vector on modern browsers. To this end,

we introduced an automated method to extract virtual function dispatches from a given binary, which we implemented in a tool called vExtractor. Furthermore, we developed a generic, static binary rewriting engine for PE files called PeBouncer that can instrument a given binary with a policy that checks the integrity of virtual function dispatches. Empirical evaluations demonstrate that our approach can detect recent zero-day vulnerabilities and the performance overhead is only slightly higher compared to compiler-based approaches.

### Acknowledgements

We thank Patrick Wollgast for porting `vtv` to MinGW-w64, Behrad Garmany and Carsten Willems for fruitful discussions, and the anonymous reviewers for their feedback. This work was supported by the German Federal Ministry of Education and Research (BMBF grant 01BY-1205A – JSAgents)

## References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] J. Afek and A. Sharabani. Dangling pointer: Smashing the pointer for fun and profit. *Black Hat USA*, 2007.

[3] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[4] Apple. Sunspider 1.0.2. `https://www.webkit.org/perf/sunspider/sunspider.html`, 2014.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[6] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, Feb. 2005.

[7] P. Becker et al. Working draft, standard for programming language c++. Technical report, Technical Report, 2011.

[8] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[9] D. Binkley and M. Harman. A Survey of Empirical Results on Program Slicing. *Advances in Computing*, 62:105–178, 2003.

[10] 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*, 2011.

[11] Botan. Botan C++ crypto library. `http://botan.randombit.net/`, 2013.

[12] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[13] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification*, pages 463–469. Springer, 2011.

[14] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[16] D. Dai Zovi. Practical return-oriented programming. *SOURCE Boston*, 2010.

[17] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[18] K. Driesen and U. Hölzle. The direct cost of virtual function calls in c++. *ACM Sigplan Notices*, 31(10):306–323, 1996.

[19] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.

[20] FireEye. Operation SnowMan. `http://goo.gl/NL0ZmV`, 2014.

[21] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, 2012.

[22] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[23] M. Harman and S. Danicic. Amorphous program slicing. In *5th International Workshop on Program Comprehension*, 1997.

[24] I. Ivanov. Api hooking revealed. *The Code Project*, 2002.

[25] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[26] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, page 1. ACM, 2014.

[27] K. D. Johnson and M. R. Miller. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities, 2010. US Patent App. 12/958,668.

[28] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[29] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `http://users.suse.com/~krahmer/no-nx.pdf`, 2005.

[30] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for java. *ACM SIGPLAN Notices*, 36(11):367–380, 2001.

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm Sigplan Notices*, 40(6):190–200, 2005.

[32] Microsoft. AppInit_DLLs in Windows 7. `http://goo.gl/BchJ4J`, 2013.

[33] Microsoft. MS13-080 addresses two vulnerabilities under limited, targeted attacks. `http://goo.gl/sCZNkL`, 2013.

[34] Microsoft. PE and COFF Specification. `http://goo.gl/EWzFcF`, 2013.

[35] Microsoft. Thiscall Calling Convention. `http://goo.gl/5o48Ub`, 2013.

[36] I. Molnar. Exec shield, new linux security feature. *News-Forge, May*, 2003.

[37] Mozilla. Firefox 0-day found on Tor .onion service. `https://bugzilla.mozilla.org/show_bug.cgi?id=901365`, 2013.

[38] Mozilla. Kraken Benchmark Suite. `http://krakenbenchmark.mozilla.org/`, 2014.

[39] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[40] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting security in cots software with binary rewriting. In *Future Challenges in Security and Privacy for Academia and Industry*, pages 154–172. Springer, 2011.

[41] S. Pichai and L. Upson. Introducing the google chrome os. *The Official Google Blog*, 2009.

[42] D. Quinlan. ROSE: Compiler Support for Object-oriented Frameworks. *Parallel Processing Letters*, 10(02/03), 2000.

[43] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib (c). In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[44] D. J. Roth and D. S. Wise. One-bit Counts Between Unique and Sticky. *SIGPLAN Not.*, 34(3), Oct. 1998.

[45] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 45–54. IEEE, 2002.

[46] V. Security. Advanced Exploitation of Mozilla Firefox Use-after-free (MFSA 2012-22). `http://goo.gl/DYB2iB`, 2012.

[47] F. J. Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[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.

[49] J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys (CSUR)*, 44(3):12, 2012.

[50] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, 2013.

[51] A. Sotirov. Heap Feng Shui in JavaScript. *Black Hat Europe*, 2007.

[52] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, technical report msr-tr-2001-50, microsoft research, 2001.

[53] B. Stroustrup. *C++*. John Wiley and Sons Ltd., 2003.

[54] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.

[55] S. Tatham, J. Hall, and H. P. Anvin. Netwide assembler, 2011.

[56] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, 2014.

[57] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[58] Y. Yang. DEP/ASLR bypass without ROP/JIT. *CanSecWest*, 2013.

[59] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy*, 2013.

[60] M. Zhang and R. Sekar. BinCFI: Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, 2013.