vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries

Aravind Prakash
Department of EECS
Syracuse University
arprakas@syr.edu

Xunchao Hu
Department of EECS
Syracuse University
xhu31@syr.edu

Heng Yin
Department of EECS
Syracuse University
heyin@syr.edu

***Abstract*—Control-Flow Integrity (CFI) is an important security property that needs to be enforced to prevent control-flow hijacking attacks. Recent attacks have demonstrated that existing CFI protections for COTS binaries are too permissive, and vulnerable to sophisticated code reusing attacks. Accounting for control flow restrictions imposed at higher levels of semantics is key to increasing CFI precision. In this paper, we aim to provide more stringent protection for virtual function calls in COTS C++ binaries by recovering C++ level semantics. To achieve this goal, we recover C++ semantics, including VTables and virtual callsites. With the extracted C++ semantics, we construct a sound CFI policy and further improve the policy precision by devising two filters, namely "Nested Call Filter" and "Calling Convention Filter". We implement a prototype system called `vfGuard`, and evaluate its accuracy, precision, effectiveness, coverage and performance overhead against a test set including complex C++ binary modules used by Internet Explorer. Our experiments show a runtime overhead of 18.3% per module. On SpiderMonkey, an open-source JavaScript engine used by Firefox, `vfGuard` generated 199 call targets per virtual callsite – within the same order of magnitude as those generated from a source code based solution. The policies constructed by `vfGuard` are sound and of higher precision when compared to state-of-the-art binary-only CFI solutions.**

I. Introduction

Control-Flow Integrity (CFI [1]) is an important program security property that needs to be ensured to prevent control-flow hijacking attacks. Because of its importance, several recent research efforts (e.g., [2]–[6]) have focused on CFI. CFI solutions roughly fall into two categories: some derive strict CFI policies from program source code (e.g., SafeDispatch [7], VTV [8]), and others such as CCFIR [2] and BinCFI [3] enforce coarse-grained CFI policy to directly protect COTS binaries. Binary-only CFI solutions are attractive, because in reality the source code for many commercial software, third-party libraries, and kernel modules are not available. However, since high-level program constructs are not necessarily preserved into the binaries during compilation, the CFI policies for these binary-only solutions are unfortunately coarse-grained and permissive.

While coarse-grained CFI solutions have significantly reduced the attack surface, recent efforts by Göktaş et al. [9] and Carlini [10] have demonstrated that coarse-grained CFI solutions are too permissive, and can be bypassed by reusing large gadgets whose starting addresses are allowed by these solutions. The primary reason for such permissiveness is the lack of higher level program semantics that introduce certain mandates on the control flow. For example, given a class inheritance, target of a virtual function dispatch in C++ must be a virtual function that the dispatching object is allowed to invoke. Similarly, target of an exception dispatch must be one of the legitimate exception handlers. Accounting for control flow restrictions imposed at higher levels of semantics is key to increasing CFI precision.

In this paper, we take a first step towards semantic-recovery-based CFI. We recover C++-level semantics to provide strict CFI protection for dynamic dispatches in C++ binaries. With protections targeting stack sanity (e.g., Stack Shield [11]), recent attacks (ab)use indirect `call` instructions to target valid function entry points, which act as large gadgets. It is essential to restrict such call targets to "legitimate" targets to prevent abuse. We set our focus on C++ binaries because, due to its object-oriented programming paradigm and high efficiency as compared to other object-oriented languages like Java, it is prevalent in many complex software programs. To support polymorphism, C++ employs a dynamic dispatch mechanism. Dynamic dispatches are predominant in C++ binaries and are executed using an indirect `call` instruction. For instance, in a large C++ binary like libmozjs.so (Firefox's Spidermonkey Javascript engine), 84.6% indirect function calls are dynamic dispatches. For a given C++ binary, we aim to construct sound and precise CFI policy for dynamic dispatches in order to reduce the space for code-reuse attacks. Our goals are similar to SafeDispatch [7] – a recent source code-based solution to protect virtual dispatches in C++ programs, but we hope to achieve the same directly on stripped binaries.

Constructing a strict CFI policy directly from C++ binaries is a challenging task. A strict CFI policy should not miss any legitimate virtual call targets to ensure zero false alarms, and should exclude as many impossible virtual call targets as possible to reduce the attack space. In order to protect real-world binaries, all these need to be accomplished under the assumption that only the binary code – without any symbol or debug information – is available. In order to construct

NDSS '15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23297

a strict CFI policy for virtual calls, we need to reliably rebuild certain C++-level semantics that persist in the stripped C++ binaries, particularly VTables and virtual callsites. Based on the extracted VTables and callsites, we can construct a basic CFI policy and further refine it. As a key contribution, we demonstrate that CFI policies with increased precision can be constructed by recovering C++-level semantics. While the refined policies may not completely eliminate code-reuse attacks, by reducing the number of available gadgets, it makes attacks harder to execute.

To evaluate our technique, we have developed a prototype called `vfGuard`. `vfGuard` is based on an open source decompilation framework [12] with over 3.4K lines of Python code. `vfGuard` successfully recovered all the legitimate VTables in complex binaries. Moreover, it recovered most of the legitimate callsites in the binary without false positives. We generated a CFI policy for each of the callsites and tested a set of C++ Windows system libraries and show under 250 call targets per binary. In comparison with BinCFI, specifically for the virtual call instructions, our policy is over 95% more precise. We show that `vfGuard` can successfully mitigate real-world exploits and report an overhead of under 20% for runtime policy enforcement.

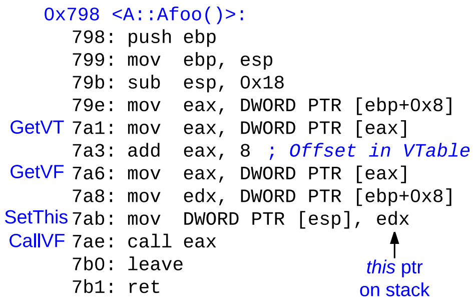
The remainder of the paper is organized as follows. Section II provides an overview of virtual methods in C++ and lessons we learned from it. Sections III provides an overview of our solution, `vfGuard`. Sections IV and V present the details of policy generation and enforcement respectively. We evaluate our solution in Section VI and present a discussion in Section VII. We present related work in Section VIII and conclude in Section IX.

II. Dynamic Dispatch in C++

In this section, using a simple running example in Figure 1 and 2, we present some background on dynamic dispatch and polymorphism in C++ from a binary perspective. Specifically, we consider two popular Application Binary Interfaces (ABIs) – Itanium [13] and MSVC [14] – that modern compilers adhere to. Figure 1(a) is the source code for a C++ program that was compiled using the g++ compiler, which adheres to Itanium ABI. Class `C` inherits from classes `A` and `B`. Figures 1(b), (c) and (e) present the layout of all the class objects, and Figure 1(d) is the layout of different VTables. A fraction of the binary code from the program is presented in Figure 2.

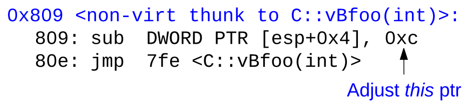
In C++, functions declared with keyword "virtual" are termed "virtual functions" [15] and their invocation is termed "virtual call" (or vcall). Virtual functions are in the heart of polymorphism, which allows a derived class to override methods in its base class. When a virtual function that is overridden in a derived class is invoked on an object, the invoked function depends on the object's type at runtime. Modern compilers – e.g., Microsoft Visual C++ (MSVC) and GNU g++ – achieve this resolution using a "Virtual Function Table" or VTable[1], a table that contains an array of "virtual function pointers" (vfptr) – pointers to virtual functions. Itanium [13] and MSVC [14] are two of the most popular C++ ABIs that dictate the implementation of various C++ language semantics.

[1]It is also called "Virtual Dispatch Table" or "Virtual Method Table".



```
0x798 <A::Afoo()>:
        798: push ebp
        799: mov  ebp, esp
        79b: sub  esp, 0x18
        79e: mov  eax, DWORD PTR [ebp+0x8]
GetVT   7a1: mov  eax, DWORD PTR [eax]
        7a3: add  eax, 8 ; Offset in VTable
GetVF   7a6: mov  eax, DWORD PTR [eax]
        7a8: mov  edx, DWORD PTR [ebp+0x8]
SetThis 7ab: mov  DWORD PTR [esp], edx
CallVF  7ae: call eax
        7b0: leave
        7b1: ret
```

(a) Assembly code for `A::Afoo`.



```
0x809 <non-virt thunk to C::vBfoo(int)>:
  809: sub  DWORD PTR [esp+0x4], 0xc
  80e: jmp  7fe <C::vBfoo(int)>
```

(b) Thunk code for `C::vBfoo`.

Fig. 2: Steps GetVT, GetVF, SetThis and CallVF are the various steps during virtual function call in `A::Afoo`. Since virtual call target `A::vAduh` does not accept any parameters, SetArg is skipped. *thunk* to `C::vBfoo` adjusts the *this* pointer before invocation.

During compilation, all the VTables used by a module are placed in a read-only section of the executable. Furthermore, a hidden field called "virtual table pointer" (vptr) – a pointer to the VTable – is inserted into objects of classes that either directly define virtual functions or inherit from classes that define virtual functions. Under normal circumstances, the vptr is typically initialized during construction of the object.

A. Virtual Call Dispatch

Figure 2(a) is marked with steps corresponding to invocation of a vcall within `A::Afoo` in Class `A` in Figure 1(a). A virtual call dispatch comprises of the following 5 steps:

GetVT	Dereference the vptr of the object (*this* pointer) to obtain the VTable.
GetVF	Dereference (VTable + *offset*) to retrieve the vfptr to the method being invoked.
SetArg	Set the arguments to the function on the stack or in the registers depending on the calling convention.
SetThis	Set the implicit *this* pointer either on stack or in `ecx` register depending on the calling convention.
CallVF	Invoke the vfptr using an indirect `call` instruction.

GetVT, GetVF, SetThis and CallVF are required steps in all vcalls, whereas depending on if the callee function accepts arguments or not, SetArg is optional. In Figure 2(a), because `A::vAduh` does not accept any arguments, SetArg is omitted. Though there is no restriction with respect to relative ordering of the steps followed, some steps are implicitly dependent on others (e.g., GetVF must occur after GetVT).

While it is usual that events GetVT through CallVF (SetArg being optional) not only occur in order, but also occur within the same basic block, it is possible – depending on the compiler

```
class A {
  int varA;
 public:
 virtual int vAfoo(int a, int *b)
{return a+(*b);}
  virtual int vAbar(int a) {return a+1;}
  virtual bool vAduh() {return true;}
  virtual int vAtest(int a) {return 0;}
  void Afoo() {this->vAduh();} };
class B {
 int varB;
 public:
  virtual int vBfoo(int a) = 0;
  virtual bool vBbar(int b)
{return b == 0;}
  char* Bfoo(char *c) {return c;} };
class C : public A, public B {
 int varC;
 public:
  int vAtest(int a) {return -(a);}
  int vAfoo(int a, int *b) {return *b;}
  int vBfoo(int a) {return a-1;}
  virtual void vCfoo() {}
  bool vAduh() {return false;}
};
```

(a)

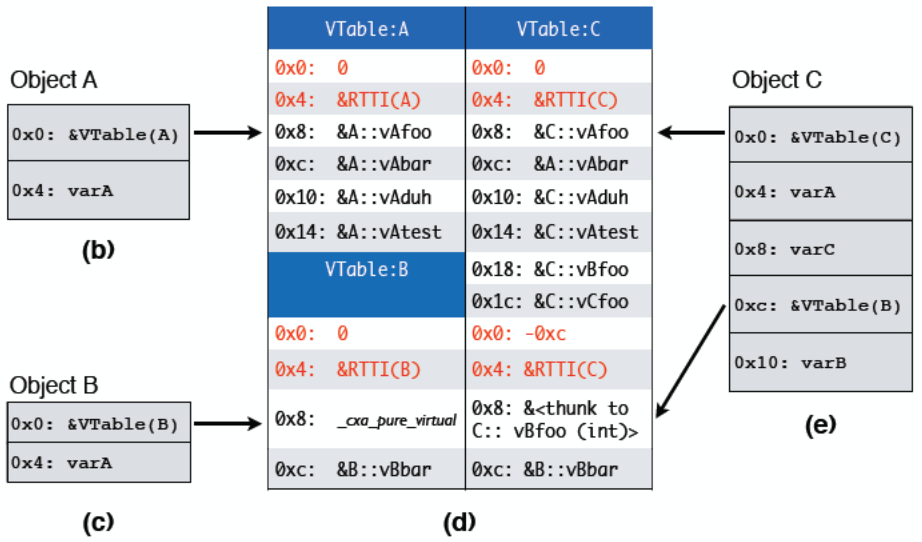


Fig. 1: (a) Source code of a sample C++ program. (b), (c) and (e) represent the object layouts of classes `A`, `B` and `C` respectively, and (d) presents the various class VTables when compiled using `g++`. Arrows point to the *address point* within the VTable. Function pointers are positioned such that the polymorphic functions are at identical offsets. E.g., offsets of `vAfoo` is 0x8 in VTables for `A` and `C`.

and the code complexity – that they occur in multiple basic blocks.

B. C++ Object and VTable Layout

Figures 1(b), (c) and (e) show object layouts for object of classes `A`, `B` and `C` respectively. An object contains various instance variables along with the vptr. Due to its frequent use, modern compilers place vptr as the first entry within the object. The vptr is followed by the member variables of the class. The location in the VTable where the vptr points to is called the "address point". The first vfptr in the VTable is stored at the address point.

In addition to an array of virtual function pointers, a VTable also holds optional information at negative offsets from address point. Optional information includes Run Time Type Information (RTTI) and various offset fields required to adjust the *this* pointer at runtime[2]. Figure 1(c) describes the VTable layout for classes `A`, `B` and `C` for C++ program in Figure 1(a). `C` inherits from `A` and `B`, therefore `C` contains two VTables, one for each base class. Since `A` comes first in the order of declaration, `C` shares its own VTable with `A`. `C`'s VTable contains all virtual functions in `A` and `B` replaced by polymorphs in `C`. Virtual functions declared by `C` follow the VTable that it shares with `A` (e.g., `C::vCfoo` in Figure 1(d)). Furthermore, since an object of `C` does not share its base address with `B` subobject within `C`, any virtual function in `B` that `C` overrides is replaced by a "thunk", which adjusts the *this* pointer to point to `C` before invoking the overridden function. For example, in Figure 2(b), `<non-virtual thunk to C::vBfoo(int)>` first subtracts the offset of `B` subobject within `C` (0xc) to reach `C` object, then invokes `C::vBfoo`.

C. Inheritance Relationship

C++ supports single, multiple, and virtual inheritance. In single and multiple inheritances, a class derives from one and more than one class respectively. Each derived class object contains all the base class objects within it. For example, Object `C` in Figure 1(e) contains both `A` and `B` objects within it. In the case of virtual inheritance, multiple base classes of a class inherit from the same base class. That is, in the running example, if `A` and `B` have a common base class "`S`" (say), then since `C` inherits from `A` and `B`, `C` would contain multiple copies of `S`, which may be undesirable. When a base class is declared virtual, the compiler maintains only one copy of `S` within `C`.

In each case of inheritance, a class shares its VTable with the first class in inheritance declaration order and contains a VTable for each subsequent base class[3]. For example, in Figure 1(e), `C` and `A` subobject of `C` start at 0x0 and share the VTable (vptr at 0x0) whereas `B` subobject of `C` starts at 0xc and contains a separate VTable accessible through vptr at 0xc. When an instance method in `B` or overridden polymorph in `C` is invoked on `C` object, before passing to the callee, the compiler adds 0xc to the *this* pointer to point to `B` subobject of `C`. Similarly, the compiler utilizes a combination of adjustment to the *this* pointer at the callsite and *thunk* to manage virtual inheritance. More details can be found in the ABI documents [13], [14].

Furthermore, when an overridden non-primary[4] base class

[2]Itanium ABI mandates RTTI and "OffsetToTop" fields. The value for RTTI is optional and is only required for dynamic type resolution through `dynamic_cast` or `type_info`. We refer readers to [13] for more information on RTTI and various offset fields.

[3]An exception is when a class has *only* virtual bases. More details can be found in the Itanium ABI document.

[4]In Figure 1(a), `B` is the non-primary base of `C` because `B` subobject in `C` does not share its address with `C` object.

virtual function is invoked in the derived class, the compiler introduces a *thunk* that adjusts the *this* pointer to point to the derived class object before invoking the actual function. For example, in Figure 2(b), the *thunk* subtracts 0xc from the `B` subobject to point to `C` object before invoking `C::vBfoo`. We refer the reader to the ABIs for more details on *thunks* and inheritance.

D. Calling Conventions

During invocation of a virtual call, the caller must pass the *this* pointer to the callee function. Two calling conventions – `stdcall` and `thiscall` [16] – are used to pass the *this* pointer to the callee. However, `thiscall` is exclusive to the MSVC compiler. In `stdcall`, the *this* pointer is passed as an implicit argument on the top of the stack, whereas, in the `thiscall` calling convention, the *this* pointer is passed through the `ecx` register. While MSVC prefers `thiscall` over `stdcall`, it defaults to `stdcall` for some functons (e.g., if it callee accepts variable arguments).

E. RunTime Type Information (RTTI)

C++ language allows runtime object type resolution. To make such resolution possible, for each class, an ABI-defined type information structure called "RunTime Type Information" is created and a pointer to the structure is stored in the VTable for the class. If a class has multiple VTables, each of the VTables contains a pointer to the same RTTI[5] structure. For example, in Figure 1(d), both VTables for class `C` contain RTTI type for `C` at offset 0x4. While such information can be very useful in reverse engineering and type reconstruction, it is an optional feature that is only required if `dynamic_cast` and/or `type_info` operators are used in the program, and is often absent in commercial software.

Based on the above description, we can make several key observations:

Ob1: VTables are present in the read-only sections of the binary.
Ob2: Offset of vfptr within a VTable is a constant and is statically determinable at the invocation callsite.
Ob3: Since the caller must pass the *this* pointer, any two polymorphic functions must adhere to the same calling convention (e.g., `C::vATest` and `A::vATest`).

III. Solution Overview

A. Problem Statement

Given a C++ binary, we aim to construct a CFI policy to protect its virtual function calls (or dynamic dispatches). Specifically, for each virtual callsite in the binary, we need to collect a whitelist of legitimate call targets. If a call target beyond the whitelist is observed during the execution of the C++ binary, we treat it as a violation against our CFI policy and stop the program execution.

[5]If the program is compiled with `-fno-rtti` flag, the RTTI field is present but contains 0.

More formally, this CFI policy can be considered as a function:

$$\mathcal{P} = \mathcal{C} \to 2^{\mathcal{F}},$$

where $\mathcal{C}$ denotes all virtual function call sites and $\mathcal{F}$ all legitimate call targets inside the given C++ binary. Therefore, as a power set of $\mathcal{F}$, $2^{\mathcal{F}}$ denotes a space of all subsets of $\mathcal{F}$. Furthermore, we define callsite $c \in \mathcal{C}$ to be a 2-tuple, $c =$ *(displacement, offset)* with displacement from the base of the binary to the callsite and the VTable offset at the callsite.

A good CFI policy must be *sound* and as *precise* as possible. The existing binary-only CFI solutions (e.g., BinCFI, CCFIR, etc.) ensure soundness, but are imprecise, and therefore expose considerable attack space to sophisticated code-reuse attacks [9]. Therefore, to provide strong protection for virtual function calls in C++ binaries, our CFI policy must be sound, and at the same time, be more precise than the existing binary-only CFI protections.

To measure the precision, we can use source-code based solutions as reference systems. With source code, these solutions can precisely identify the virtual dispatch callsites and the class inheritance hierarchy within the program. Then, at each callsite, they insert checks to ensure that (1) the VTable used to make the call is compatible with the type of the object making the call [8] or (2) the call target belongs to a set of polymorphic functions extracted from the inheritance tree for the type of object making the call [7].

Assumptions and Scope. Since we target COTS C++ binaries, we must assume that none of source code, full symbol information, debugging information, RTTI, etc. is available. We must also deal with challenges arising due to compiler optimizations that blur and remove C++ semantic information during compilation. In other words, we must rely on strong C++ semantics that are dictated by C++ ABIs and persist during the process of code compilation and optimization. Due to the reliance on standard ABIs, we only target C++ binaries that are compiled using standard C++ compilers (e.g., MSVC and GNU g++). Custom compilers that do not adhere to Itanium and MSVC ABIs are out of scope. Moreover, since our goal is to protect benign C++ binaries, code obfuscation techniques that deliberately attempt to evade and confuse our defense are also out of our scope.

Furthermore, our goal is to protect virtual function calls and their manifestations through indirect `call` instructions in the binary. We do not aim to protect indirect `jmp` or `ret` instructions. However, we aim to provide a solution orthogonal to existing solutions (e.g., shadow call stack [11], coarse-grained CFI [2], [3]) so as to provide a more complete and accurate CFI.

B. Our Solution

In order to tackle the problem stated in Section III-A, we must leverage the C++ ABIs to recover strong C++ semantics that persist in a given C++ binary. First of all, we need to accurately discover virtual callsites $\mathcal{C}$ in the binary. Then, we need to identify all the virtual function entry points, which form the legitimate call targets $\mathcal{F}$. Because all functions in $\mathcal{F}$ are polymorphic (virtual), and must exist in VTables, we must

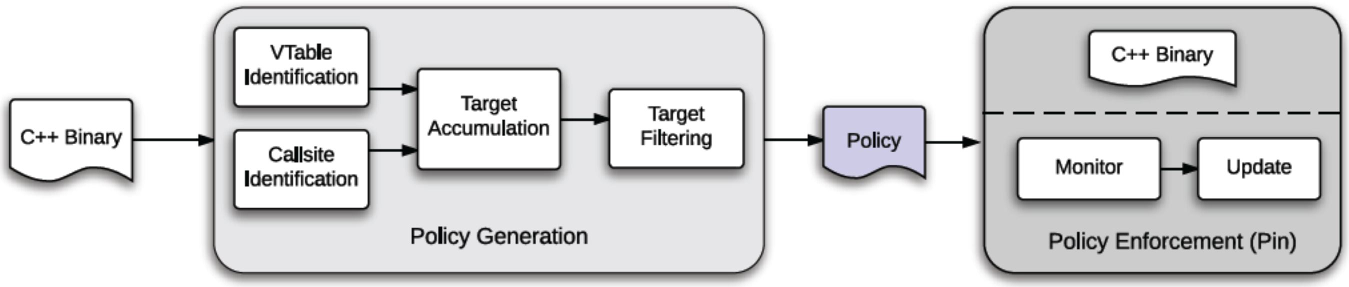


Fig. 3: Overview of `vfGuard`. It consists of Policy Generation and Policy Enforcement stages.

identify all the VTables in the binary. After having identified virtual callsites and VTables, we can construct a basic policy such that, for each callsite the legitimate targets include all the functions in the VTables at the given offset. This basic policy is already more precise than the state-of-the-art CFI defenses, CCFIR and BinCFI. To further improve the policy precision, we propose two additional filters to reduce this set of legitimate targets. Once the policy is generated, we can instrument the C++ binary to enforce the generated policy.

Figure 3 presents the overview of our solution – `vfGuard`. Given a C++ binary, `vfGuard` will extract virtual callsites in the "Callsite Identification" component, and VTables in the "VTable Identification" component. With the extracted callsites and the VTables, the "Target Accumulation" component accumulates all the functions from the VTables that individual callsites can target. Finally, "Target Filtering" filters the targets to obtain a more precise policy. While the generated policy can be enforced in multiple ways, as a proof-of-concept, we provide a runtime enforcement to hightlight the feasibility of our solution. In the remainder of this section, we provide a brief overview of individual components of `vfGuard`.

C. Callsite Identification

The key challenge in identifying the callsites is to differentiate legitimate callsites from other indirect call instructions. C++ binary is often mixed with non-C++ code components written in C, assembly language, etc. These code components could be included from a dependent library or could be injected by the compiler for exception handling, runtime binding, etc. While the steps enumerated in Section II-A provide a starting point to look for callsites within a binary, they present some hard challenges:

- Some steps are independent of the other and therefore follow no strict ordering – e.g., SetArg and SetThis may occur before GetVT and GetVF.
- GetVT through CallVF may span multiple basic blocks – e.g., when two or more virtual calls are dispatched in multiple branches on the same *this* pointer, a compiler may move some steps to a basic block that dominates the blocks that performs the calls.
- Some steps may be implicit. For example, in functions that employ `thiscall` convention, an incoming object pointer may be retained in the `ecx` register throughout the function thereby eliminating the need for an explicit SetThis for virtual calls on the same object.
- GetVF resembles a simple dereference if offset is 0. This bears close resemblance to a double dereference of a function pointer in C code. To ensure soundness in our policy generation, we cannot afford to include false virtual callsites.

To address all the above challenges, we take a principled approach and perform static and flow-sensitive intra-procedural data flow analysis on an intermediate representation of the binary. Being flow-sensitive helps in mitigating complications occurring due to invocation steps being carried out in different basic blocks. Moreover, x86 binary code is complex and is made up of hundreds of non-intuitive instructions. It offers multiple instructions to accomplish a given task. For example, moving the VTable address into a register can either be accomplished using a `mov` instruction or an `lea` instruction. By transforming the binary to semantic-preserving, yet simple intermediate representation, we greatly simplify data flow analysis. This also lets us handle complexities arising due to invocation of virtual functions at offset 0 within VTables.

In summary, in each function that contains a potential callsite (i.e., indirect call instruction), we transform the binary into an intermediate representation and represent the callsite as a set of simplified expressions comprising of input variables to the function. For legitimate callsites, these expressions represent the steps carried out in calling a virtual function. From the expressions, we extract the offset and the symbolic object pointer that is used to make the virtual call. SetThis is a key step in legitimate virtual call dispatch and – in significant cases – it is a deciding factor in identifying a legitimate callsite. `vfGuard` performs "Object Pointer Analysis" to detect SetThis, wherein, it tracks assignment to registers to capture the assignment of the implicit *this* pointer. More details will be provided in Section IV-A.

D. VTable Identification

Reconstructing precise inheritance tree from the binary is ideal but hard. For instance, Dewey et al. [17] locate the constructors in the program by tracking the VTable initializations. In commercial software, constructors are often inlined, therefore such an approach would not yield a complete set of VTables that we seek. Other approaches use heuristics that are not only dependent on debug information, but also tailored for specific compilers like MSVC (e.g., IDA VTBL plugin [18]).

We propose an ABI-centric algorithm that can effectively recover *all* the VTables in a binary in both MSVC and Itanium ABIs. Based on **Ob1** in Section II, we scan the read-only sections in the binary to identify the VTables. The VTables that adhere to Itanium ABI contain mandatory fields along with the array of vfpts. The mandatory fields make locating of VTables in the binary relatively easier when compared to MSVC ABI. However, VTables generated by the Microsoft Visual Studio compiler (MSVC ABI) are often grouped together with no apparent "gap" between them. This poses a challenge to accurately identify VTable boundaries.

Furthermore, according to **Ob2**, we first scan the code and data sections and identify all the "immediate" values. Then, we check each value for a valid VTable address point. A valid VTable contains an array of one or more vfptrs starting from the address point. It is possible that our algorithm identifies non-VTables – e.g., function tables that resemble VTables – as genuine VTables. We err on the safe side, because including a few false VTables does not compromise the policy soundness and only reduces precision to a certain degree. A detailed algorithm for VTable identification is presented in Section IV-B.

E. Target Accumulation and Filtering

All the vfptrs and *thunks* within all the VTables together form a universal set for virtual call targets. A naive policy will include *all* vfptrs as valid targets for each callsite. For large binaries, such a policy would contain 1000s of targets per callsite. While still more precise than existing defenses, it would still expose a large attack space. We leverage the offset information at the callsite to obtain a more precise policy.

Given an offset at a callsite, during "Target Accumulation", we obtain a basic policy for each callsite that encompasses all the vfptrs (and *thunks*) at the given offset in all the VTables in which the offset is valid. Additionally, we apply two filters to further improve the policy precision. First, we note that target vfptrs for a callsite that is invoked on the same object pointer as the host function must belong to the same VTables as the host function. With this, we apply our first filter, called "Nested Call Filter". Furthermore, from **Ob 3**, the calling convention that is presumed at the callsite and the calling convention adhered to by the target function must be compatible. Accordingly, we apply the second filter called "Calling Convention Filter".

We considered several other filters, but did not adopt them for various reasons. To name a few, we could infer the number of arguments accepted by each function and require it to match the number of arguments passed at the callsite; we could perform inter-procedural data flow analysis to keep track of *this* pointers; and we could perform type inference on function parameters and bind type compatible functions together.

$function$	$::=$	$(stmt)*$
$stmt$	$::=$	$var ::= exp \mid exp ::= exp \mid \texttt{goto}\ exp$
		$\mid \texttt{call}\ exp \mid \texttt{return}$
		$\mid \texttt{if}\ var\ \texttt{then}\ stmt$
exp	$::=$	$exp\ \Diamond_b\ exp \mid \Diamond_u exp \mid var$
$\Diamond_b$	$::=$	$=, +, -, *, /, ...$
$\Diamond_u$	$::=$	$deref, -, \sim$
var	$::=$	$\tau_{reg} \mid \tau_{val}$
τ_{reg}	$::=$	$\texttt{reg1\_t} \mid \texttt{reg8\_t} \mid \texttt{reg16\_t} \mid \texttt{reg32\_t}$
τ_{val}	$::=$	$\{Integer\}$

TABLE I: Intermediate Language used by `vfGuard`. *deref* corresponds to the dereference operation.

However, at binary level, such analyses are imprecise and incomplete. A function may not always use all the arguments declared in source code, and thus we may not reliably obtain the argument information in the binary. Inter-procedural data flow analysis and type inference are computational expensive, and by far not practical for large binaries. We will investigate more advanced filters as future work.

F. Policy Enforcement

To enforce the generated policy, we need to perform binary instrumentation, which can be either static or dynamic. Several dynamic binary instrumentation tools (e.g., Pin [19] and DynamoRIO [20]) are publicly available and are fairly mature. Therefore, we build our policy enforcement based on Pin. We did experiment with several static instrumentation tools such as IDA Pro [21] and Pebil [22]. However, these tools have not been robust enough to deal with complex large C++ binaries. Nevertheless, since our main contribution lies in policy generation, we do not believe that our contribution will be diminished without static binary instrumentation.

Although the policy enforcement is straightforward, we still need to deal with several practical issues. It is possible for modules to inherit from classes that are defined in other modules. However, the policy generated by `vfGuard` is applicable within a module, so when a virtual dispatch invokes a function in another module, the invocation violates the policy. To tackle this problem, we maintain a list of modules that a program depends on, then at runtime, the policies pertaining to a module are updated based on the load address of the module and the policies are accordingly merged. More details about enforcement is presented in Section V.

IV. Policy Generation

Policy generation comprises of first identifying the legitimate callsites and the VTables in the binary, then based on the offset at the callsite and the VTables, a basic policy is generated for each callsite. Two filters are applied to further refine the targets.

A. Callsite Identification

Due to the complex nature of x86 binaries and the complexities involved in recovering the callsites, a simple scanning-based approach for callsite identification is insufficient. `vfGuard` first identifies all the candidate functions that could host callsites by identifying functions in the binary that contain at least 1 indirect call instruction. Each identified function is subjected to static intra-procedural analysis to identify legitimate virtual callsites and VTable offsets at such callsites. In order to perform the data flow analysis, we modified an open source `C` decompiler [12]. Below, we present the different steps in our analysis.

IR Transformation and SSA Form. x86 instruction set is large, and instructions often have complex semantics. To aid in analysis and focus on the data flow, we make use of a simple, yet intuitive intermediate language as shown in Table I. The IR is simple enough to precisely capture the flow of data within a function without introducing unnecessary overhead. Each function is first broken down into basic blocks and a control-flow graph (CFG) is generated. Then, starting from the function entry point, the basic blocks are traversed in a depth-first fashion and each assembly instruction is converted into one or more IR statements. A statement comprises of expressions, which at any point is a symbolic representation of data within a variable. A special unary operator called $deref$ represents the dereference operation. `goto, call` and `return` instructions are retained with similar semantic interpretations as their x86 counterparts. Edges between basic blocks in the CFG is captured using `goto`.

In its current form, `vfGuard` supports registers up to 32 bits in size, however, the technique itself is flexible and can be easily extended to support 64-bit registers. Moreover, note that the ABIs are not restricted to any particular hardware architecture. Performing analysis on the IR facilitates our solution to be readily ported to protect C++ binaries on other architectures (e.g., ARM) by simply translating the instructions to IR.

Furthermore, we convert each IR statement into Single Static Assignment (SSA) [23], [24] form, which has some unique advantages. IR in SSA form readily provides the *def-use* and the *use-def* chains for various variables and expressions in the IR.

Def-Use Propagation. The definition of each SSA variable and list of statements that use them constitutes the Def-Use chains [23]. `vfGuard` recursively propagates the definitions into uses until all the statements are comprised of entry point definitions (i.e., function arguments, input registers and globals). Due to flow-sensitive nature of our analysis, it is possible that upon propagation, we end up with multiple expressions for each SSA variable, and each expression represents a particular code path. For example, for the class hierarchy in Figure 1(a), consider the code snippet:

```
   ...
1. A *pa; A a; C c;
2. if (x == 0)
3.    pa = &a
4. else
5.    pa = &c
6. pa->vAtest(0);
   ...
```

At line 6, depending on the value of `x`, the vfptr corresponding to `vAtest` could either be `&(&(&c)+0x14`[6]`)` or `&(&(&a)+0x14)`. Assuming `stdcall` convention, per step SetThis, the implicit object pointer could either be `&c` or `&a`. Precise data flow analysis should capture both possibilities, and for each case ensure the existence of a corresponding *this* pointer assignment on the stack. For such cases, `vfGuard` creates multiple copies of the statement – one for each propagated expression.

Subsequently, each definition is recursively propagated to the uses until a fixed point is reached. At each instance of propagation, the resulting expression is simplified through constant propagation. For example, $deref((ecx_0 + c_1) + c_2)$ becomes $deref(ecx_0 + c_3)$ where $c_3 = c_1 + c_2$.

Callsite Identification. As per the steps involved in dynamic dispatch described in Section II-A, we need to capture GetVT through CallVF using static data flow analysis. More specifically, for each indirect call, we compute expressions for the call target and expressions for *this* pointer passed to the target function. Note that due to flow-sensitive data flow analysis, we may end up having multiple expressions for each statement or variable.

For a virtual callsite, after def-use propagation, its call instruction must be in one of the two forms:

$$\texttt{call}\ deref(deref(exp) + \tau_{val}) \tag{1}$$

or

$$\texttt{call}\ deref(deref(exp)) \tag{2}$$

In the first form, exp as an expression refers to the vptr within an C++ object and τ_{val} as a constant integer holds the VTable offset. When a virtual callsite invokes a virtual function at offset 0 within the VTable, the call instruction will appear in the second form, which is a double dereference of vptr. Here, τ_{val} is the byte offset within the VTable and must be divisible by the pointer size. Therefore, if τ_{val} is not divisible by 4, the callsite is discarded.

Next, we need to compute an expression for *this* pointer at the callsite. *this* pointer can be either passed through `ecx` in `thiscall` or pushed onto the stack as the first argument in `stdcall` conventions. Expression for *this* pointer must be identical to the exp within the form (1) or (2).

Table II presents a concrete example. At 0x7ae, after propagation and simplification, the call instruction matches with form (1) and we determine the expression for *this* pointer to be $deref(esp_0 + 4)$ and VTable offset as 8. Then at 0x7ab, we determine that the first argument pushed on the stack is also $deref(esp_0+4)$. Now, we are certain that this callsite is indeed a virtual callsite, and it uses `stdcall` calling convention.

It is worth noting that our technique is independent of the inheritance structure and works not only for single inheritance, but also multiple and virtual inheritances. This is because the compiler adjusts *this* pointer at the callsite to point to appropriate base object *before* the virtual function call is invoked. Therefore, while the expression for *this* pointer may vary, it must be of the form (1) or (2) above. Our method aims

[6]Note that the offset for `vAtest` from Figure 1(d) is 0x14.

Address	Instruction	IR-SSA form	After Propagation and Constant Folding
0x798	`push` ebp	$deref(esp_0) = ebp_0$ $esp_1 = esp_0 - 4$	$deref(esp_0) = ebp_0$ $esp_1 = esp_0 - 4$
0x799	`mov` ebp, esp	$ebp_1 = esp_1$	$ebp_1 = esp_0 - 4$
0x79b	`sub` $esp, 0x18h$	$esp_2 = esp_1 - 0x18$	$esp_2 = esp_0 - 0x1C$
0x79e	`mov` $eax, [ebp + 8]$	$eax_0 = deref(ebp_1 + 8)$	$eax_0 = deref(esp_0 + 4)$
0x7a1	`mov` $eax, [eax]$	$eax_1 = deref(eax_0)$	$eax_1 = deref(deref(esp_0 + 4))$
0x7a3	`add` $eax, 8$	$eax_2 = eax_1 + 8$	$eax_2 = deref(deref(esp_0 + 4)) + 8$
0x7a6	`mov` $eax, [eax]$	$eax_3 = deref(eax_2)$	$eax_3 = deref(deref(deref(esp_0 + 4)) + 8)$
0x7a8	`mov` $edx, [ebp + 8]$	$edx_0 = deref(ebp_1 + 8)$	$edx_0 = deref(esp_0 + 4)$
0x7ab	`mov` $[esp], edx$	$deref(esp_2) = edx_0$	$deref(esp_2) = deref(esp_0 + 4)$
0x7ae	`call` eax	`call` eax_3	`call` $deref(deref(deref(esp_0 + 4)) + 8)$

TABLE II: Callsite identification for A::Afoo() in the sample example in Figure 2(a). After propagation at 0x7ae, $deref(esp_0+4)$ is the *this* pointer and 8 is the VTable offset.

to resolve *this* pointer directly for each callsite, and thus can deal with all these cases.

B. VTable Identification

`vfGuard` scans read-only sections in the C++ binary for VTables, using a signature template derived from the Itanium and MSVC ABI specifications. Moreover, since the VTable locations are known during compile-time, using `mov` or an equivalent instruction, compilers move the "immediate" value of a VTable's address point into the object base address during object initialization. Therefore, VTable base addresses are a subset of all the "immediate" values that occur in the binary. `vfGuard` first scans for all the "immediate" values that occur in the binary. Then, the value is identified as a VTable if it belongs to a read-only section and matches with the signature template. A VTable is considered valid if it contains an array of one or more vfptrs at the address point. Figures 1(d) and 4 present the VTables for Itanium and MSVC ABIs. Under Itanium ABI, the RTTI information and *Offset-to-top* – offset from the subobject (`B` in `C`) to base object (`C`) are contained as mandatory fields within the VTable and therefore provide a stronger signature template. They also act as a natural boundary between VTables. MSVC imposes no such requirement. In binaries compiled by MSVC, groups of VTables are often contiguously allocated, thereby presenting the challenge of identifying the boundaries.

The algorithm used to identify VTables is presented in Algorithm 1. It comprises of two functions. "ScanVTables" takes a binary as input and returns a list of all the VTables $\mathcal{V}$ in the binary. Each instruction in the code sections and each address in the data sections of the binary are scanned for immediate values. If an immediate value that belongs to a read-only section of the binary is encountered, it is checked for validity using "getVTable" and $\mathcal{V}$ is updated accordingly. "getVTable" checks and returns the VTable at a given address. Starting from the address, it accumulates entries as valid vfptrs as long as they point to a valid instruction boundary within the code region. Note that not all valid vfptrs may point to a function entry point. For instance, in case of "pure virtual" functions, the vfptr points to a compiler generated stub that jumps to a predefined location. In fact the MSVC compiler introduces stubs consisting of a single return instruction to implement empty functions. To be conservative, `vfGuard` allows a vfptr to point to any valid instruction in the code segments. Upon failure, it returns the accumulated list of vfptrs as the VTable entries. If no valid vfptrs are found, an empty set – signifying invalid VTable address point – is returned.

Furthermore, the following restriction is imposed on Itanium ABI: A valid VTable in the Itanium ABI must have valid RTTI and "OffsetToTop" fields at negative offsets from the address point. The RTTI field is either 0 or points to a valid RTTI structure. Similarly, "OffsetToTop" must also have a sane value. A value -0xffffff $\leq$ *offset* $\leq$ 0xffffff, which corresponds to an offset of 10M within an object, was empirically found to be sufficient. Depending on the specific classes and inheritance, fields like "vbaseOffset" and "vcallOffset" may be present in the VTable. To be conservative, we do not rely on such optional fields. However with stronger analysis and object layout recovery, these restrictions can be leveraged for more precise VTable discovery.

While `vfGuard` may identify some false VTables as legitimate, its conservative approach does not miss a legitimate VTable, which is a core requirement to avoid false positives during enforcement. Moreover, Algorithm 1 is not very effective at detecting end points of the VTables in the binary (e.g., in Figure 4, end points of all VTables would be captured as 0x74). Pruning the VTables based on neighboring VTable start addresses (e.g., since `B` starts from 0x50, setting A's end to be 0x4c) may lead to unsound policies if the neighboring VTables are not legitimate. While our approach reduces precision, it keeps the policy sound. Our algorithm terminates a VTable when the vfptr is an invalid code pointer.

While Itanium ABI provides strong signatures for VTables due to mandatory offsets, MSVC ABI does not. In theory, any pointer to code can be classified as a VTable under MSVC ABI. In practice however, we found that legitimate VTables contain at least 2 or more entries. Therefore, under MSVC ABI, we consider VTables only if they contain at least 2 entries.

C. Target Filtering

Basic Policy. Based on the identified callsites and the VTables V, `vfGuard` generates a basic policy. For a given callsite c with byte offset o, we define index k to be the index within

Algorithm 1 Algorithm to scan for VTables.

```
procedure getVTable($Addr$)
    $V_{methods} \leftarrow \emptyset$
    if ($ABI_{Itanium}$ and $isMandatoryFieldsValid(Addr)$) or $ABI_{MSVC}$ then
        $M \leftarrow [Addr]$
        while $isValidAddrInCode(M)$ do
            $V_{methods} \leftarrow V_{methods} \cup M$
            $Addr \leftarrow Addr + size(PTR)$
            $M \leftarrow [Addr]$
        end while
    end if
    return $V_{methods}$
end procedure

procedure ScanVTables($Bin$)
    $\mathcal{V} \leftarrow \emptyset$
    for each $Insn \in Bin.code$ do
        if $Insn$ contains $ImmediateVal$ then
            $C \leftarrow immValAt(Insn)$
            if $C \in Section_{RO}$ and $getVTable(C) \neq \emptyset$ then
                $\mathcal{V} \leftarrow \mathcal{V} \cup C$
            end if
        end if
    end for
    for each $Addr \in Bin.data$ do
        if $[Addr] \in Section_{RO}$ and $getVTable([Addr]) \neq \emptyset$ then
            $\mathcal{V} \leftarrow \mathcal{V} \cup [Addr]$
        end if
    end for
    return $\mathcal{V}$
end procedure
```

VTable:A	VTable:C
0x40: &A::vAfoo	0x58: &C::vAfoo
0x44: &A::vAbar	0x5c: &A::vAbar
0x48: &A::vAduh	0x60: &C::vAduh
0x4c: &A::vAtest	0x64: &C::vAtest
VTable:B	0x68: &C::vBfoo
	0x6c: &C::vCfoo
0x50: _purecall	0x70: &<thunk to C::vBfoo(int)>
0x54: &B::vBbar	0x74: &B::vBbar

Fig. 4: VTables with mandatory fields for running examples as generated by Visual Studio (MSVC) compiler (no RTTI). `A::Afoo` and `B::Bfoo` are non-virtual member functions and therefore are not present in the VTables. VTables for `A`, `B` and `C` are aligned one after the other. It is important to identify them as separate VTables and not a single large VTable.

the VTable that the byte offset corresponds to (i.e., k = o/4 for 4-bytes wide pointers). The legitimate call targets of c must belong to a subset of all the functions at index k within in all the VTables that contain at least $(k + 1)$ vfptrs. Here we assume VTables to be zero-based arrays of vfptrs. That is:

$$Targets = \{V_i[k] \mid V_i \in \mathcal{V},\ |V_i| > k\},$$

where V_i is the VTable address point. $|V_i|$ is the number of vfptrs in the VTable at V_i.

In Figure 4, since class `C` overrides function `vBfoo` in base class `B`, the compiler introduces a *thunk* to perform runtime adjustments (Section II). An astute reader may wonder if *thunk* could cause a problem. The answer is no. Since `vBfoo` is not present in the VTable for `B`, it is not a valid target for offset 0. Note that this is the expected behavior since `vBfoo` is a pure virtual function that cannot be directly invoked. However, *thunk* is a valid VTable entry and therefore captured as a valid target for offset 0.

Nested Virtual Call Filtering. In some cases, *this* pointer used to invoke a virtual function is later used to make one or more virtual calls within the function body. We refer to such virtual calls as "Nested Virtual Calls". `vfGuard` can generate a more precise policy for nested virtual calls.

```
class M { virtual void vMfoo() {
    //vFn(); or
    this->vFn(); } };
```

In the above example, `vFn` is a virtual function that is invoked on the same *this* pointer as its host function `M::vMfoo`, which is also a virtual function. Underneath, the binary implementation reuses the VTable used to invoke `M::vMfoo` to retrieve the vfptr (or *thunk*) pertaining to `vFn`. That is, between the nested virtual calls and the host virtual function, the vptr acts an invariant. Therefore at the nested callsite, target `vFn` must belong to a VTable to which `M::vfoo` also belongs.

Given a virtual callsite with vfptr index k and host virtual function f, we can derive a more precise policy for each nested virtual callsite within f:

$$Targets = \{V_i[k] \mid V_i \in \mathcal{V}, f \in V_i\}$$

Nested virtual callsites can be easily identified using our intra-procedural data flow analysis. First, we check whether the *this* pointer at the given callsite is in fact the *this* pointer for the host function. That is, the expression for *this* pointer at that callsite should be ecx_0 for `thiscall` calling convention or $esp_0 + 4$ for `stdcall` calling convention. Next, we ensure that the host function is virtual. That is, there must exist at least 1 VTable to which the host function belongs. Finally, the filtered targets are identified using the equation above.

Note that the filter is applicable only in cases where host function is also virtual. For example, in Figure 1(a) (and Table II), `A::Afoo` reuses the *this* pointer to invoke `vAduh`, a virtual function. Since `A::Afoo` is not a virtual function, it is not contained within any VTable and therefore, $Targets$ will be $\emptyset$. If the filter is inapplicable, `vfGuard` defaults to basic policy.

Calling-Convention based Filtering. We filter the target list to be compatible with the calling convention followed at the callsite. At the callsite, the register that is utilized to pass the implicit *this* pointer (CallVF) reveals the calling convention that the callee function adheres to.

First, for each of the callsite target functions in the policy, we identify the calling convention the function adheres to. Next, for each callsite, we check if there is a mismatch between the convention at the callsite and the target, if so, we remove such conflicting targets from the list. If we are unable to identify the calling convention of the callee – which is possible if the callee does not use the implicit *this* pointer (e.g., `A::vAbar, A::vAfoo`, etc.), we take a conservative approach and retain the target.

Incomplete Argument Utilization. Conceptually, all polymorphs of a function must accept the same number of arguments. So, one potential filter could be to check if a function accepts the same number of arguments that are passed at the callsite. If not, the mismatching functions can be removed from potential targets for the callsite. However, in the binary, we only see the number of arguments *used* by a function and not the number of arguments it *can* accept.

Mismatch for legitimate targets arises when a base class and a derived class operate on unequal number of arguments for a given virtual function. For example, in Figure 1(a), we identify 3 arguments (2 arguments + *this*) for `A::vAfoo` whereas only 2 arguments (1 argument + *this*) for `C::vAfoo`. Reverse is the case for `A::vAtest` and `C::vAtest` with 1 and 2 arguments repectively. For a callsite that invokes `vAtest`, depending on the type of the object, both `A::vAtest` and `C::vAtest` are legitimate targets. Therefore, argument count is *not* feasible as a filter.

V. Policy Enforcement

We enforce the policy generated by `vfGuard` by running the program on Pin: a dynamic binary translator. Other techniques such as Binary Rewriting([25] and [22]) as used by [1]–[3], in-memory enforcement by injecting code into process memory (e.g., using Browser Helper Objects [26]), etc. are equally feasible.

In our proof-of-concept approach, we intercept the control flow at every previously identified callsite and check if the call target is allowable for the callsite. If the target is dis-allowed, the instance is recorded as a violation of policy. Effective enforcement must impose low space and runtime overheads. Under the basic policy, `vfGuard` captures the policy within 2 maps. The first map M_{cs} maps a callsite to the VTable offset at the callsite, and the second map M_{target} maps a given target to a 160-bitvector[7] that represents the valid VTable offsets for the given target. That is, i^{th} bit set to 1 indicates that the target is valid for offset $i*4$. For a given CS, a M_{cs} entry is readily derived from τ_{val} in Equation 1 in Section IV-A. M_{target} is populated from the identified VTables (Section IV-B). For each VTable entry, the corresponding bitvector is updated to indicate a 1 for the offset at which the entry exists within the VTable. `vfGuard` performs 2 map lookups and 1 bitvector masking to verify the legitimacy of a given target at a callsite. That is, for a given callsite (CS) and target (T), the target is validated if:

$$BitMask(M_{cs}(CS)) \ \& \ M_{target}(T) \neq 0 \tag{3}$$

[7]The size of the bitvector is dictated by the size of the largest VTable in the binary. We found 160 to be sufficient.

Such a design enables quick lookup and limits space overhead from duplication of callsites and targets within the maps. While the map lookups and bitvector masking result in constant time runtime overhead, the space requirements of M_{cs} and M_{target} are linear with resepct to number of callsites and targets respectively.

In case of callsites whose targets were filtered, the target lookup is different from basic policy. Each callsite CS – whose targets were filtered – is associated with a map $M_{Filtered}(CS)$ that maintains all the allowable call targets for CS. During enforcement, `vfGuard` first checks if the callsite is present in $M_{Filtered}(CS)$ and validates the target. If callsite is not present in $M_{Filtered}$ (i.e., targets for the callsite were not filtered), `vfGuard` performs the 2 map lookup and verifies the target through Equation 3. Enforcing the filtered policy introduces greater space overhead. The main reason being: targets reappear in multiple $M_{Filtered}$ for each of the callsites that the targets are valid at. We wish to investigate better enforcement in our future work.

Effect of ASLR:. `vfGuard` performs policy enforcement with or without ASLR enabled. The callsite and targets in M_{cs}, M_{target} and $M_{Filtered}$ are stored as (module, offset) tuple rather than the concrete virtual address. When a module is loaded, the virtual addresses of callsite and target addresses are computed from the load address of the module.

A. Cross-Module Inheritance

In practice, classes in one module can inherit from classes defined in another module [27]. Therefore, as new modules are loaded into a process address space, the allowable call targets for callsites in existing modules need to evolve to accommodate the potential targets in the new module. Given a list of approved modules that a program depends on, `vfGuard` can analyze each of the modules to generate the intra-module policy. From the execution monitor, `vfGuard` monitors module loads to capture any newly loaded modules and their load addresses. Intra-module policies are progressively adjusted (for ASLR) and maps M_{cs}, M_{target} and $M_{Filtered}$ are updated so as to capture the allowable targets for various callsite offsets across all approved modules. If a target in an unapproved modules is invoked, `vfGuard` records it as a violation.

B. Other Enforcement Strategies

Policy enforcement performed by `vfGuard` is only a proof-of-concept and not the focus. Prior approaches (e.g., [1]–[3]) have leveraged static instrumentation to introduce Inline Reference Monitors (IRMs) to check the legitimacy of a branch target at runtime. We believe such approaches can improve the performance of `vfGuard`. Furthermore, depending on the number of modules loaded, the size of callsite and target maps can increase to result in significant memory overhead, specially in case of filtered targets. In such cases, cross-module dependencies can be analyzed to only allow cross-module calls in cases where known dependencies exist, thereby controlling the size of various enforcement maps.

VI. Implementation and Evaluation

We implemented `vfGuard` in the following code modules. The policy generation part of `vfGuard` is implemented as a

Program	Ground Truth	vfGuard	FP	FN
SpiderMonkey	811	942	13.9%	0
dplus-browser_0.5b	270	334	19.1%	0
TortoiseProc.exe	568	595	4.7%	0

TABLE III: VTable Identification accuracy.

Program	Ground Truth	vfGuard	FP	FN
SpiderMonkey	1780	1754	0	1.4%
dplus-browser_0.5b	309	287	0	7.1%

TABLE IV: Callsite Identification Accuracy. Ground-truth was generated using gcc with `-fvtable-verify` compile option.

plugin for IDA-pro v6.2. An open source IDA-decompiler [12] was modified to perform data flow analysis for callsite identification. The platform consists of 5.6K lines of Python code and 3.4K lines were added to it. A PinTool [19] was written using 850 lines of C++ code to perform policy enforcement.

We evaluated `vfGuard` in several respects. We first evaluated the accuracy of virtual callsite and VTable identification using several open source C++ programs, because source code is needed to obtain ground truth. Then, we measured the policy precision and compare with BinCFI and SafeDispatch. To evaluate the effectiveness of `vfGuard`, we tested multiple realworld exploits. Finally, we measured `vfGuard`'s coverage with respect to number of indirect branches protected, and performance overhead of policy enforcement.

Test Set. To evaluate `vfGuard`, we consider a set of C++ program modules presented in Tables III, IV and V. Firstly, our test set comprises of programs containing between 100 and 2000 VTables, thereby providing sufficient complexity for analysis. Secondly, the modules in the test set are a part of popular browsers like Firefox and Internet Explorer, which are known to contain several vulnerabilities. SpiderMonkey is the JavaScript engine employed by FireFox and Table V presents some of the modules used by Internet Explorer that contain reported vulnerabilities. Finally, the test set contains both open (Table III, IV) and closed source programs (Table V). While `vfGuard` operates on raw COTS binaries, open source programs provide the ground truth to evaluate `vfGuard`'s accuracy. Along with SpiderMonkey and the modules used by IE, the set consists of dplus browser, an open source browser and TortoiseSVN, an open source Apache subversion client for Windows.

A. Identification Accuracy

To ensure policy soundness, `vfGuard` must identify all legitimate VTables and must not identify any false virtual callsites. To measure the accuracy, we picked SpiderMonkey and dplus-browser for the Itanium ABI, and TortoiseProc for the MSVC ABI. We constructed the "ground truth" by using compiler options that dump the VTables and their layouts in the binary. Specifically, `-fdump-class-hierarchy` and `/d1reportAllClassLayout` compiler options were used to compile the programs on g++ and Visual Studio 2013 respectively. The results are tabulated in Table III. The compilers emit meta-data for (1) each class object's layout in the memory, and (2) each VTable's structure. We compared each of the VTables obtained from the ground truth against `vfGuard`. None of the legitimate VTables were missed in each of the cases. In all the cases, VTables identified by `vfGuard` contained some noise (from 4.7% to 19%). This was expected due to the conservative nature of `vfGuard`'s VTable scanning algorithm.

To evaluate callsite identification accuracy of `vfGuard`, we leveraged a recent g++ compiler option, `-fvtable-verify` [8] that embeds checks at all the virtual callsites in the binary to validate the VTable that is invoking the virtual call. We compiled SpiderMonkey with and without the checks, and matched each of the callsites that contained the compiler check to the callsites identified by `vfGuard`. Out of the functions that were successfully analyzed, `vfGuard` reported 0 false positives. It reported 1.4% and 7.1% false negatives (i.e., missed during identification) for SpiderMonkey and dplus-browser respectively.

These experiments indicate that the generated policies should be sound but a little imprecise, due to the noisy VTables and missing callsites.

B. Policy Precision

To measure how precise our generated policies are, we generate policies for C++ binary modules in Internet Explorer 8. Table V presents the average number of targets per callsite under 3 configurations – basic policy, basic policy with Nested Callsite Filter (NCF) and basic policy with Nested Callsite Filter and Calling Convention Fiilter (CCF). Additionally, for each case, we estimated the number of targets in a policy generated by BinCFI. We included into the policy all the function entry points in the program. The exact reduction in the number of targets is tabulated in the last column. We can see that even with the basic policy generated by `vfGuard`, we were able to refine BinCFI's policy by over 95%. Here, the refinement numbers pertain to the virtual callsites protected by `vfGuard` and not all the indirect branch instructions within the module. An optimal defense will combine `vfGuard`'s policy for virtual callsites along with those generated by BinCFI (or CCFIR) for other branch instructions.

In general, we found no obvious correlation between the number of callsites and VTables in the binary to the effectiveness of the filters. While the filters improved precision in some cases, they did not in others. Graphs in Figure 5 show the scattered distribution of number of callsites with respect to offset within the VTable at the callsite, and the number of VTables that contain a particular VTable size for `mshtml.dll` and `wmvcore.dll`. As expected, we found several VTables with small sizes of less than 10 elements. However, we also found a significant number of VTables between sizes 50 and 125. While the algorithm used by `vfGuard` is efficient in detecting the VTable start address, it is not very accurate in detecting the end points. This is the main hurdle for lowering the average call targets per callsite.

Program	Total VTables Identified	Total Callsites Identified (CS)	Avg. Targets per CS (Basic Policy)	# Nested CS	Avg. Targets per CS (NCF)	Avg. Targets per CS (NCF+CCF)	Estimated `call` Targets – BinCFI	Call Target Reduction w.r.t BinCFI
ExplorerFrame.dll	736	6314	231	257	227	223	8964	97.5%
msxml3.dll	587	3321	96	219	88	84	6822	98.8%
jscript.dll	129	1170	39	55	38	38	2314	98.4%
mshtml.dll	1174	3583	292	211	258	257	16287	98.3%
WMVCore.dll	736	7516	268	562	256	244	8845	97.3%

TABLE V: Average targets for the basic policy and the filters. NCF is the Nested Callsite Filter and CCF is the Calling Convention Filter. Nested Callsites is the number of callsites that share the object pointer with the host.

Program	Total # Indirect `call` instructions	Total # Indirect `jmp` instructions	Total # `ret` instructions	Total # Indirect `calls` analyzed (instructions successfully transformed to IR)	% of analyzed `calls` protected	% of Total indirect `calls` protected
ExplorerFrame.dll	7797	87	7266	7042	89.7%	81.0%
msxml3.dll	5439	78	6157	4045	82.1%	61.1%
jscript.dll	2235	5	4430	1678	69.7%	52.3%
mshtml.dll	9843	352	15479	4598	77.9%	36.4%
WMVCore.dll	9748	50	8497	8223	91.4%	77.1%

TABLE VI: Profile of indirect branch instructions and `vfGuard` policy coverage in the modules tested.

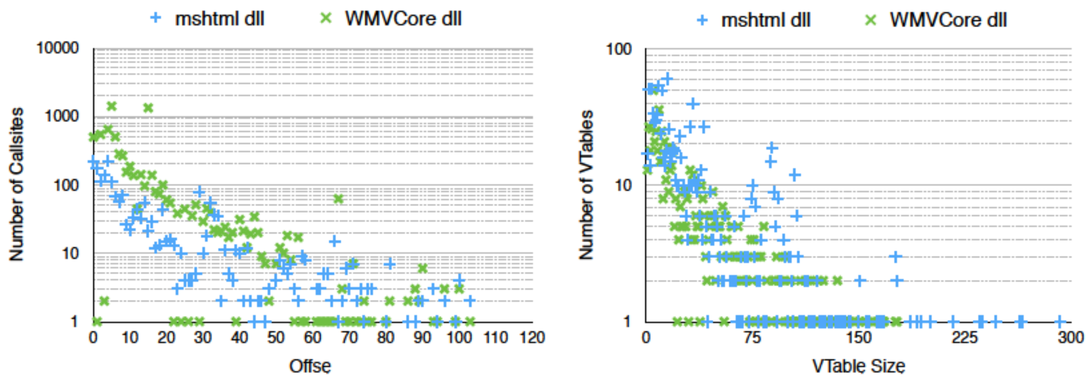


Fig. 5: Distribution of callsites across various offsets for mshtml.dll and WMVCore.dll.

We also want to evaluate the precision of the policy generated by `vfGuard` as compared to SafeDispatch, a source code based solution. To estimate the policy produced by SafeDispatch, we modified g++ compiler option `-fvtable-verify` implementation. For every invoked callsite, it will insert code to output the number of possible targets. We compiled SpiderMonkey with this modified g++ compiler and ran its test cases. We observed that the average number of call targets per callsite is 109, and the maximum is 335. In comparison, `vfGuard` generated a policy for SpiderMonkey with 199 call targets per callsite on average and a maximum of 943 targets. This result indicates that the precision of CFI policies generated by `vfGuard` is within the same order of magnitude as those generated from a source code based solution.

C. Policy Effectiveness

To assess the policy effectiveness, we conducted a survey on VTable related exploits. In particular, we found five working exploits towards Internet Explorer and Firefox, and listed them in Table VII. We tested the two exploits that target `mshtml.dll`. Being protected with the generated policy for `mshtml.dll`, Internet Explorer was able to detect these exploits successfully.

CVE	Target Application	Module	Remark
2010-0249	Internet Explorer	mshtml.dll	Fake VTable
2013-1690	Firefox	xul.dll	Fake VTable
2011-1255	Internet Explorer	mshtml.dll	Fake VTable
2010-3962	Internet Explorer	mshtml.dll	Mis-aligned VTable access
2013-3893	Internet Explorer	mshtml.dll	Fake VTable

TABLE VII: Exploit mitigation. VTable based vulnerabilities.

In CVE-2010-0249, an attacker sprays the heap with fake VTables and corrupts a stale object pointer by setting its vptr to the heap sprayed region. Then, at a callsite inside `mshtml.dll!CElement::GetDocPtr()`, attacker controlled vfptr is retrieved and executed. Since the attacker supplied vfptr is not a part of the policy for the callsite, it is flagged by `vfGuard` as an exploit.

In the case of CVE-2010-3962, `mshtml.dll` inadvertently increments the vptr of an object. So, in the virtual dispatch in `CLayout::EnsureDispNodeBackground()`, [address-point + `offset` + 1] is retrieved as the address of the vfptr instead of [address-point + `offset`]. Since the misaligned pointer does not belong to the policy, `vfGuard` flags it as an exploit. While we tested the above exploits successfully, more exploits in Table VII follow the same modus operandi.

D. *Policy Coverage*

Exploits in the wild utilize indirect branch instructions that include indirect `call`, `jmp` and `ret` instructions to subvert flow of control. Scope of `vfGuard` is limited to protecting the virtual function callsites. Table VI presents the profile of indirect instructions in various modules tested under MSVC ABI along with coverage achieved by `vfGuard`. `vfGuard` leverages the modified version of IDA-decompiler [12] – an open source decompiler – as a platform for IR transformation. Due to the complexity and use of unsupported instructions (e.g., floating point instructions), the transformation to IR failed for certain functions. "Total # Indirect calls analyzed" is the number of indirect call instructions in the functions that were successfully converted to IR. The problem was particularly severe in `mshtml.dll` because the platform failed to analyze 485/4515 functions that contain indirect `call` instructions. Those 485 functions were not only complex, but also accounted for a significant fraction of indirect calls. On average, `vfGuard` was able to protect 82.16% of all the indirect calls that were successfully transformed to IR. This accounted for 61.58% of all the indirect calls – including non-virtual dispatch callsites – in all the modules. While the remaining 38.42% of indirect `call` instructions are still vulnerable to code-reuse attacks involving large gadgets (as demonstrated by [9]), we believe that `vfGuard` provides significant reduction in attack space. With more complete IR transformation, we expect more indirect calls to be protected.

Furthermore, Table VI lists indirect branch instructions in each of the modules. Indirect `call` and `ret` instructions dominate indirect branching. 45.3% (average) of all the indirect branch instructions in the test set resulted from indirect `call` instructions.

E. *Performance Overhead*

`vfGuard` performs policy enforcement using PinTool, a publicly available process-level runtime execution monitor. To measure the performance overhead imposed by `vfGuard`, we opened load-intensive webpages on Internet Explorer and recorded the overhead on 3 individual modules with respect to Pin as baseline. The results are graphed in Figure 6. Overall, we found an average overhead of 18.3%. We made no attempt to optimize the runtime performance of policy enforcement. While this performance overhead is not impressively low, it is aligned with other binary-level CFI protection solutions, such as BinCFI.

VII. Discussion

How to better identify VTable end points. The strictness of the policy or the attack space depends on the number of call-targets per callsite. Ideally, we want this number to be as close to the ideal case as possible. However, inaccurate VTable end-points – specially in MSVC ABI – result in inclusion of incorrect vfptrs into the policy.

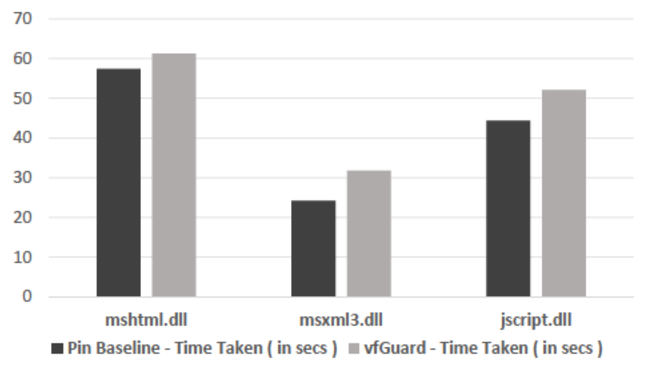


Fig. 6: Performance overhead imposed by `vfGuard` in comparison with Pin as baseline.

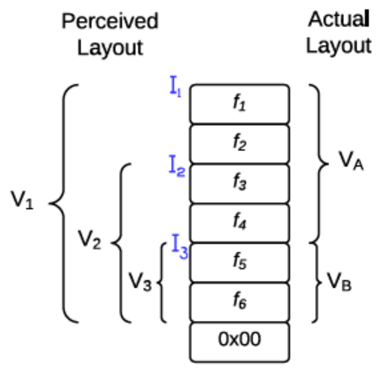


Fig. 7: Actual and perceived VTable layouts under MSVC ABI.

Consider the layout in Figure 7. V_A and V_B are 2 VTables under MSVC ABI that are contiguously allocated, where V_A is comprised of entries f_{1-4} and V_B of entries f_{5-6}. I_{1-3} are addresses within the function pointer array that manifest within the binary as immediate values, where I_1 and I_3 are VTable address points and I_2 is noise. Per Algorithm 1, `vfGuard` identifies 3 VTables V_{1-3} with a total of 6+4+2=12 entries. Accordingly (due to V_2) f_3, f_4, f_5 and f_6 are incorrectly included in the policy for offsets 0, 1, 2 and 3 respectively, thereby compromising precision over soundness. Similarly, f_5 and f_6 are allowed as legitimate targets for offsets 4 and 5 respectively.

One solution to such a problem would be to prune VTables based on the start addresses of succeeding VTables in the memory. However, such a solution must have no false positives in VTable start-addresses. In Figure 7, since I_2 is an incorrect VTable address point, f_{3-4} would be incorrectly excluded from V_1 thereby leading to false positives during enforcement. Another solution could leverage more restrictions from the

ABI and language semantics to better demarcate VTables. For example, colocated functions at a given offset – e.g., `A::vAfoo` and `C::vAfoo` at offset 0x8 in Figure 1(d) – must be compatible with each other with respect to types of arguments accepted and type of value returned. We intend to pursue this direction in our future work.

Virtual-dispatch-like C calls. Our virtual callsite identification has captured all required steps for a virtual dispatch according to C++ ABI specifications, but it is still possible that some functions in the C code could resemble a legitimate C++ virtual function dispatch. For example:

```
pa->pb->fn(pa);
```

In the binary, the above C statement resembles a C++ virtual call dispatch. `pa` being passed as an argument satisfies SetThis and could be perceived as a callsite. Commercial compilers tend to follow a finite number of code patterns during a virtual call invocation. A potential solution could classify all the callsites based on code patterns used to perform the dispatch and look for abnormalities. For example, to dispatch virtual calls in mshtml.dll, the compiler typically invokes virtual calls using a call instruction of the form, "`call` [*reg* + *offset*]" where as, g++ produces code that performs, "`add` *reg, offset*; `call` *reg*". While this is not a standard, a compiler tends to use similar code fragments to dispatch virtual calls within a given module. Since a sound policy must prevent false callsites, one can filter the potential incorrect callsites by looking for persistent virtual dispatch code fragments.

VIII. Related Work

A. Control Flow Integrity

Control flow integrity was first proposed by Abadi et al. in 2005 [1] as part of the compiler framework to automatically place in-line reference monitors in the emitted binary code to ensure the legitimacy of control transfers. Since then, a great deal of research efforts have built on top of it. Some efforts extended the compiler framework to provide better CFI protection. In particular, MCFI [5] enables a concept of modular control flow integrity by supporting separating compilation. KCoFI [28] provides control flow integrity for commodity operating system kernels. RockJIT [4] aims to provide control flow integrity for JIT compilers.

Other efforts are made to enforce control flow integrity directly on binary code. Efforts such as PittSFIeld [29] and CCFIR [2] enforce coarse-grained policy by aligning code. Based on a CPU emulator, Total-CFI [30] provides CFI protection for multiple processes running in an entire operating system by extracting legitimate jump targets from relocation tables and import tables. MoCFI [6] rewrites ARM binary code to retrofit CFI protection on smartphone.

B. VTable-Based Analysis

Security solutions have leveraged VTables in C++ programs both with and without source code. VTV [8] and SafeDispatch [7] provide compiler-based solutions to analyze various callsites in C++ source code and instrument virtual function calls to perform runtime checks to validate VTable or target virtual function for each callsite. VTGuard [31] inserts cookies into VTables at compile-time and at runtime, verifies the cookie at the callsite to ensure legal VTable use.

Callsite identification in our work is similar to T-VIP [32]: work done independently and concurrently with ours. T-VIP starts from an indirect `call` instruction, obtains a backward slice and analyzes the slice to identify various steps in a virtual call. They instrument instructions that load VTable to validate integrity of VTable. Along similar lines, VTint [33] instruments C++ binaries to enforce a policy that requires VTables to originate from read-only sections. `vfGuard` identifies the allowable call targets at callsites to generate a precise CFI policy. It offers protection at the callsite by restricting targets as opposed to ensuring VTable integrity.

C. Binary Reverse Engineering

Reverse engineering data structures from binary executables is very valuable for many security problems. TIE [34] and Rewards [35] make use of dynamic binary analysis to recover the type information and data structure definitions from the execution of a binary program. On the other hand, recovering C++ level semantics and data structures poses a different set of problems. It is generally considered hard primarily due to the underlying complexity in the language semantics when compared to C.

The problem of C++ reverse engineering was revisited by Sabanal et al. [36] in 2007. They use heuristic based scanning approach to identify different C++ code constructs within a binary and recover C++ level semantics. More recently, Folkin et al. [37] take a principled approach based on the C++ language properties. They target to reconstruct the C++ class hierarchy from the binary. Objdigger [38] tracks definition and uses of *this* pointer through inter-procedural data flow analysis to identify various objects instances and member methods of a class. While these solutions are interested in recovering the precise C++ semantics, our approach is more tuned in leveraging the restrictions imposed by such semantics.

D. C++ De-compilation

Several efforts have been made in the direction of C++ decompilation [37], [39], [40]. A key task these tools must accomplish is to recover the class hierarchy from the binary. Our goals are significantly different. We take an ABI-driven platform agnostic approach to infer the restrictions that C++ semantics impose on the binary. While recovery of the class hierarchy improves our precesion, it is not a requirement. Availability of RTTI information and other debug information can only improve the quality of defense provided by `vfGuard`.

IX. Conclusion

In this paper, we present `vfGuard`, a system to generate a strict CFI policy for virtual calls in C++ binaries. It performs static analysis on the C++ binary directly to extract virtual callsites and VTables reliably to ensure the soundness of the generated policy and achieve high precision at the same time. According to our experiments with realworld C++ binaries, we demonstrated that `vfGuard` can indeed generate a sound CFI policy for protecting virtual calls. For the callsites protected by vfGuard, it can achieve over 97% target reduction, compared to BinCFI. Even under suboptimal enforcement using Pin, vfGuard showed an average overhead of 18.3% per module.

ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their comments. We would also like to thank Pallavi Iyengar, Masters student at Syracuse University for helping us evaluate vfGuard. This research was supported in part by NSF Grant #1018217, NSF Grant #1054605 and DARPA Cyber Grand Challenge. Any opinions, findings and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, 2005, pp. 340–353.

[2] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland'13)*, 2013, pp. 559–573.

[3] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 22nd USENIX Security Symposium (Usenix Security'13)*, 2013, pp. 337–352.

[4] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of 21st ACM Conference on Computer and Communication Security (CCS '14)*, 2014.

[5] ——, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[6] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-r. Sadeghi, "MoCFI: A framework to mitigate control-flow attacks on smartphones," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, 2012.

[7] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ virtual calls from memory corruption attacks," in *Proceedings of 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[8] 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 *Proceedings of 23rd USENIX Security Symposium (USENIX Security'14)*, 2014, pp. 941–955.

[9] E. Göktaş, E. Anthanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of 35th IEEE Symposium on Security and Privacy (Oakland'14)*, 2014.

[10] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[11] "Stack Shield," http://www.angelfire.com/sk/stackshield/.

[12] F. Chagnon, "IDA-Decompiler," https://github.com/EiNSTeiN-/ida-decompiler.

[13] "Itanium C++ ABI," http://mentorembedded.github.io/cxx-abi/abi.html.

[14] J. Ray, "C++: Under the hood," http://www.openrce.org/articles/files/jangrayhood.pdf, March 1994.

[15] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley, 2013.

[16] "THISCALL calling convention," http://msdn.microsoft.com/en-us/library/ek8tkfbw.aspx, 2013.

[17] D. Dewey and J. T. Giffin, "Static detection of C++ vtable escape vulnerabilities in binary code." in *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, 2012.

[18] Nektra, "Vtbl – IDA plugin," https://github.com/nektra/vtbl-ida-pro-plugin, 2013.

[19] 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," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, 2005, pp. 190–200.

[20] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000, pp. 1–12.

[21] "The IDA Pro disassembler and debugger," https://www.hex-rays.com/products/ida/.

[22] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for linux," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*, March 2010.

[23] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[24] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.

[25] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary transformation in a distributed environment," Microsoft Research, Tech. Rep. MSR-TR-2001-50, April 2001.

[26] "Browser Helper Objects," http://sysinfo.org/bhoinfo.html.

[27] "Using dllimport and dllexport in C++ classes," http://msdn.microsoft.com/en-us/library/81h27t8c.aspx.

[28] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Proceedings of 35th IEEE Symposium on Security and Privacy (Oakland'14)*, 2014.

[29] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture," in *Proceedings of the 15th Annual USENIX Security Symposium (Usenix Security'06)*, 2006.

[30] A. Prakash, H. Yin, and Z. Liang, "Enforcing system-wide control flow integrity for exploit detection and diagnosis," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS'13)*, 2013, pp. 311–322.

[31] M. Miller and K. Johnson, "Using virtual table protections to prevent the exploitation of object corruption vulnerabilities," Patent, Jun. 7, 2012, US Patent App. 12/958,668. [Online]. Available: http://www.google.com/patents/US20120144480

[32] R. Gawlik and T. Holz, "Towards automated integrity protection of C++ virtual function tables in binary programs," in *Proceedings of 30th Annual Computer Security Applications Conference (ACSAC'14)*, Dec 2014.

[33] C. Zhang, C. Song, Z. K. Chen, Z. Chen, and D. Song, "VTint: defending virtual function tables integrity," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[34] J. H. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[35] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, 2010.

[36] P. V. Sabanal and M. V. Yason, "Reversing C++," *Blackhat Security Conference*, 2007.

[37] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina, "Smartdec: Approaching C++ decompilation," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, 2011, pp. 347–356.

[38] 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 (PPREW'14)*, 2014, pp. 1–11.

[39] "Boomerang decompiler," http://boomerang.sourceforge.net/.

[40] I. Skochinsky, "Practical C++ decompilation." [Online]. Available: https://archive.org/details/Recon_2011_Practical_Cpp_decompilation