Program Obfuscation via ABI Debiasing

David Demicco
Computer Science
Binghamton University
United States
ddemicc1@binghamton.edu

Rukayat Erinfolami
Computer Science
Binghamton University
United States
rerinfo1@binghamton.edu

Aravind Prakash
Computer Science
Binghamton University
United States
aprakash@binghamton.edu

ABSTRACT

The Itanium ABI is the most popular C++ ABI that defines data structures essential to implement underlying object-oriented concepts in C++. Specifically, name mangling rules, object and VTable layouts, alignment, etc. are all mandated by the ABI. Adherence to the ABI comes with undesirable side effects. While it allows interoperability, past research efforts have shown that it provides robust inference points that an attacker can leverage to reveal sensitive design information through binary reverse engineering. In this work, we aim to reduce the ability of an attacker to successfully reverse engineer a binary. We do this via removal of what we call *ABI Bias*, i.e., the reverse engineering bias that manifests due to a compiler's adherence to the ABI.

Specifically, we identify two types of ABI biases that are central to past reverse engineering works on C++ binaries: VTable ordering bias and Function Pointer bias. We present compiler-based techniques that can correctly and efficiently *debias* a given binary from the aforementioned biases. We evaluate our proof-of-concept implementation on a corpus of real world programs for binary size, correctness and performance. We report an average increase of 1.42% in binary size compared to the baseline, very low performance overhead and lastly, correct execution of evaluation programs in comparison to the baseline. Finally, we demonstrate efficacy of our approach by hindering DeClassifier, a state-of-the-art C++ reverse engineering framework.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

C++, security, reverse engineering

ACM Reference Format:
David Demicco, Rukayat Erinfolami, and Aravind Prakash. 2021. Program Obfuscation via ABI Debiasing. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3485832.3488017

1 INTRODUCTION

Interfaces, particularly ABIs, are central to software re-usability and interoperability. As such, multiple factors play a role during ABI design. Ease and feasibility of implementation, performance, and compliance to existing standards are key considerations and frequently debated topics during ABI design [13].

Once standardized, ABIs must withstand the test of time and offer very little room (if any) for modification. While popular ABIs (e.g., Itanium ABI) have been successful in meeting the functional and performance goals of modern software, their inability to adapt to evolving security threats has been a problem. For instance, discussions in the mailing list archives for the cxx-abi discussion group [13] typically originate with a sample implementation and *lack discussion on security implications including reversibility of binaries.*

A binary's adherence to an ABI provides robust security-sensitive inference points for reverse engineers. There are two types of information that are of interest to a reverse engineer. On the one hand, there is the valuable program logic itself. There has been a substantial amount of work in the field of decompilation that aims at program logic recovery. However, there has also been concerted effort at obfuscation that aims to deter program logic recovery [17]. On the other hand, there is the high-level design information (e.g., class inheritance) that is revealed by the very virtue of adherence to an ABI. *In order to ensure interoperability, there can be no compromise on ABI adherance.* In fact, recent efforts [7, 23, 28] have shown that ABI adherence can indeed reveal sensitive high-level semantics. For example, until recently, Itanium ABI [1] mandated that secondary VTables of a complete-object VTable immediately follow the primary VTable. While such a mandate may seem benign and inconsequential, reverse engineers [7, 23] rely on such a strict ordering of secondary VTables to recover valuable semantics (e.g., class inheritance graph) from the binary.

In fairness to the committee on Itanium ABI standard, the strict ordering requirement has been recently relaxed (see [2] §2.5.2, last para), however *the relaxation is primarily motivated by performance and ease of implementation, and not security and/or reversibility.* Furthermore, the LLVM and GCC compilers (two most popular compilers that subscribe to the Itanium ABI) continue to emit VTables that conform to earlier versions of the ABI—i.e., secondary VTables are emitted immediately after the primary VTable of a complete-object VTable.

In this paper, we address the dichotomy that exists between a constantly changing threat model and fairly stagnant ABIs. Specifically, we define *ABI-bias* that aids in reverse engineering and present compiler techniques that can eliminate bias—*without compromising the conformance of the binary to the ABI.* We look at the C++ Itanium ABI due to its wide commercial use and the attention it gets from both attackers [4, 9, 25] and defenders [3, 14, 15, 19, 20, 22, 26, 27] alike.

ABI Bias: We present the notion of *ABI bias*, a pro-reverse-engineering bias that manifests due to a program's conformance to an ABI standard. Because these biases are inherent to ABI conformance, they are impervious to obfuscation. While obfuscators can hide program logic, conformance to ABI specification must be retained in order to preserve interoperability. We focus on two specific biases that are central to past reverse-engineering efforts on C++ binaries. First, there is the VTable ordering bias or *VTBias* that manifests due to strong guarantees provided by the ABI regarding ordering of VTables of polymorphic classes. Past efforts [6, 21, 23, 28] have relied on such an ordering to identify complete-object VTables that uniquely represent polymorphic classes in the binary. Second, there is the Function Pointer bias or *FPBias* that is a result of the number of function pointers in a VTable. The number of function pointers and their ordering reveal valuable inheritance information in the binary. Past efforts [10, 11] have relied on this primitive to establish directionality of polymorphic inheritance graphs recovered from C++ binaries.

We also present the notion of *Lingering Bias*. The idea that even if an ABI is changed, programs that where created with the older ABI can still leave important information available to reverse engineers. This can happen when a change in the ABI[1] is backwards-compatible.

We present principled compiler-based techniques to debias the Itanium ABI. By focusing on insensitive aspects of the ABI that do not impact program interoperability or execution (more in Section 3), our solution eliminates bias and poses significant hurdles to reverse engineering while preserving backward compatibility. Particularly, we implement debiasing solutions against VT bias and FP bias on the LLVM compiler while incurring near-zero runtime overhead.

We applied our solution to a wide range of real world applications with varying complexities (ranging from 944 to 14635 polymorphic classes). Our results show that the file size increase for eliminating VT bias is less than 0.1% in most cases. The file size overhead introduced by elimination of FP bias varies depending on other optimizations that are in place, however it typically stays < 2%. We found the median runtime overhead while debiasing FP bias to be < 1%. We also tested our solution against DeClassifier [7], a modern reverse engineering tool, demonstrating that it breaks the VTable groupings it relies on leading to incorrect groupings and results. Our contributions can be summarized as follows:

(1) We present the notion of *ABI bias*, a bias that aids in reverse engineering by very virtue of conformance to an ABI. We identify 2 distinct biases: VTable bias and Function-Pointer bias that arise from the ABI requirement for how VTables and functions within them must be laid out.
(2) We present an implementation based on the LLVM compiler that eliminates VT and FP biases while ensuring that backward compatibility is not lost.
(3) We evaluate our solution against a corpus of 7 real-world programs (including 2 C++ programs from SPEC 2017 suite) and demonstrate correctness, interoperability and low performance overhead.
(4) We demonstrate the efficacy of our solution against DeClassifier, a modern reverse engineering tool.

The rest of the document is organized as follows. The Section 2 provides technical background necessary to understand the remainder of the paper, Section 3 presents an overview of our approach, Section 4 and 5 present the technical details of our solution. Sections 6 and 7 present the evaluation and security analysis of our work respectively. We present the related work in Section 8 and finally conclude in Section 9.

2 BACKGROUND

2.1 Polymorphism in C++

Polymorphism is one of the features of C++ that allows functions to behave differently depending on the runtime type of the object they are invoked on. This capability can be implemented when there is inheritance, that is one or more classes derived from one or more other classes. A function in the base class can be overridden and implemented differently in the derived class. Such functions must be defined as virtual functions. A class that defines virtual function(s) is referred to as a polymorphic class. The Itanium ABI [2] defines a per-polymorphic-class structure called a VTable, which contains a list of pointers to all virtual functions of that polymorphic class in the order in which they are laid out in the source code. Function pointers are used at runtime to dispatch virtual functions. Since C++ permits a class to inherit directly from multiple classes (called multiple inheritance), a class can comprise of more than one VTable (or sub-VTables). The collection of all VTables belonging to a class is referred to as a complete-object VTable. A class shares its primary VTable with its primary base class and has secondary VTable(s) corresponding to its secondary base(s).

An object of a class will contain multiple sub-objects if it inherits from multiple bases. Like VTables, the derived class sub-object is shared with the primary base and every secondary base has a corresponding sub-object. The constructor of a class writes the primary VTable pointer (vptr) into the primary sub-object and does the same for the other sub-objects. Such patterns occur frequently enough in practice to be a concern [8].

2.2 VTable Layout

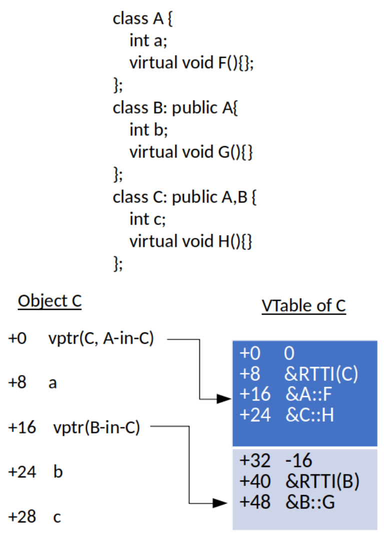
A VTable contains some mandatory fields namely: OTT (offset-to-top), RTTI (runtime type information) and one or more virtual function pointers. The OTT specifies the offset that must be added to the address of a sub-object to obtain the address of the complete (derived) object. The OTT is zero for primary VTables and a negative value for secondary VTables. The RTTI points to a structure that contains the class hierarchy information of a given class. Specifically, it contains pointers to the RTTI of the base classes in the order in which they occur in the class hierarchy. It is useful for performing `dynamic_cast` to verify at runtime if the type an object is to be cast is valid. RTTIs are generated only for polymorphic classes. If RTTI is disabled (using `-fno-rtti` flag on g++), the RTTI field of the VTable contains value zero. Lastly the virtual function pointer fields point to virtual functions.

The primary VTable of a class contains the pointers to all the functions defined by the primary base class (with overridden functions replaced with pointers to the overriding functions), followed by the functions defined by the derived class including overridden functions of secondary base(s). A secondary VTable contains pointers to functions defined in the secondary base, with overridden functions replaced with “thunks" that perform necessary adjustment to the *this* pointer before transferring control to the overriding function in the primary VTable.

2.3 Object Memory Layout and VTable Pointers

The first item in a newly constructed polymorphic object is a VTable pointer. The pointer to the object is referred to as the *this* pointer, and it points to the VTable that is associated with this class in memory, giving it access to the function pointers that are members of the class. The VTable pointer is then followed by any other data members the object contains, initialized (or not) by the constructor during construction.

In the case of inheritance however, the constructor has a bit more to do, and the layout in memory can get more complex. Consider the simple case seen in Figure 1. The object consists of a pointer to C’s VTable, followed by the data members of A, then a second pointer to the VTable of B::C, followed by B’s data members, and finally c’s data members. The ordering and layout of the VTable is specified by the ABI.



```
class A {
   int a;
   virtual void F(){};
};
class B: public A{
   int b;
   virtual void G(){}
};
class C: public A,B {
   int c;
   virtual void H(){}
};
```

Figure 1: Simple VTable layout in memory, showing VTable Pointers.

2.4 ABI-Based Semantics Inference

The ABI’s specification on the implementation of certain aspects of the C++ language provides robust means of inferring semantics such as class hierarchy from the binary. Some existing hierarchy recovery tools use analysis techniques that are based on such specifications.

VTable Scanning and Grouping. Almost all reverse engineering efforts start by excavating VTables from binaries through a scanning process, also known as VTable scanning (see Table 1, column VTSc). First, distinct VTable signatures are derived from the ABI (based on layout and mandatory fields), then a static analysis approach is employed to scan the read-only sections of the binary and excavate VTables. Because the signatures are robust and guaranteed by the compiler, they are relatively easy to identify following a set of heuristics. First, VTables contain sensitive data in the form of function pointers, and so must not be altered by program code during runtime. As such, they are always allocated in read-only sections of the binary. Second, the layout of a VTable is fixed. It always begins with an OTT, followed by an RTTI value, followed by some number of function pointers, and then any number of sub-VTables, which follow the same layout. Third, these values all have patterns and restrictions on what they can be. The OTT must be zero for the primary VTable, and must decrease in all the following sub-VTables. The RTTI field must be a valid pointer or 0. Worth noting is that it is not necessary to check the information the RTTI entry points to - in fact this can be detrimental - as some programs simply do not have this information available for use. The function pointers must all point to the beginning of a function or thunk in the executable sections of the program, and so cannot point to data. In some edge cases (caused by pure virtual functions and abstract classes), the function pointers may point to an entry corresponding to `__purevirtual` exception handling mechanism provided by the C++ runtime. Most VTable scanning techniques account for these edge cases in their heuristics.

From the scanned and identified VTables, VTable grouping can then identify complete-object VTables that uniquely represent a polymorphic class. According to the Itanium ABI, the primary VTable of the derived class is followed by the secondary VTables of its base classes (non-virtual bases before virtual bases). vfGuard [23] and DeClassifier [7] use this information to group the VTables associated to a class into the complete object VTable. This is done by first sorting VTables in increasing order of VTable addresses, then grouping a primary VTable (with zero offset-to-top value) with succeeding zero or more secondary VTables (with a negative OTT value). VTable grouping is useful to identify the distinct number of classes present in the binary. This also helps to build a clear and concise class hierarchy graph where each node is a complete object VTable.

VTable Size. The ABI mandates that a derived class VTable contains all entries of the base class VTable (with appropriate replacements in case functions are overridden) along with additional entries introduced by the derived class. Therefore, the size of a derived class VTable is at least equal to the sum of the sizes of its base class(es) VTables.

Table 1: Binary level defenses and their adopted techniques. VTSc: VTable scanning, VTG: VTable grouping, OA: Overwrite analysis, VTS: VTable size, CC: Constructor call.

	Techniques adopted				
Defense	VTSc	VTG	OA	VTS	CC
DeClassifier [7]	✓	✓	✓	✓	✓
vfGuard [23]	✓	✓			
Marx [21]	✓		✓		
ROCK [18]	✓			✓	✓
SmartDec [10]	✓			✓	✓
VCI [6]	✓				✓

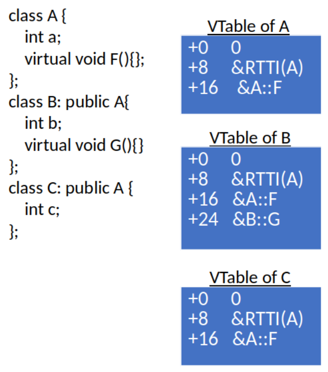


Figure 2: Simple VTable layouts demonstrating their contents and size.

For a clear example of how this works, consider the simplest case shown in figure 2. The base class A has a total VTable size of 24 bytes, containing the offset to top, RTTI, and the address of function &A::F (F in A). However class B, derived from A, also contains the address of the function &B::G (G in B) making its size 32 bytes. Should another class derive from B, it would also at least contain B's function pointers and A's function pointer. Class C in this case, does inherit from A, however it adds no function pointer of its own, and so does not increase the size of the VTable, (this is the case where the complete object VTable is equal in size to the the base's complete object VTable). Class D in Figure 3 has multiple inheritance, and so has a sub-VTable as well as its primary VTable. This sub-VTable makes the objects size 56 bytes, bigger than objects of class A or B.

While the ABI does not specify that additional information cannot be added to VTables, the implication is strong enough that ROCK [18] and SmartDec [10] take advantage of VTable sizes to assign direction of inheritance to related classes and to eliminate impossible base classes respectively.

Constructor Analysis. The ABI specifies the object construction process wherein base class sub-objects are recursively constructed before constructing the derived object. It is a popular primitive among past efforts (Table 1 column "CC") to infer inheritance order. However, compilers are known to aggressively inline constructors, and therefore constructor analysis is not a reliable source of inheritance order.

3 ABI BIAS

Typical binary reverse-engineering work-flow emanates from revealing factors that are discernible in the binary. For example, calls to libc functions (e.g., `strcpy`) in a binary provide type information regarding input variables to the function and therefore provide a basis for further analysis (e.g., backward slicing). Similarly, in C++ reverse engineering, a complete-object VTable provides a clear and unambiguous representation of a polymorphic class.

We define *ABI Bias* as an ABI property that results in *unavoidable* revelation of forensically-relevant information by a binary due to the very virtue of it adhering to an ABI. For example, until recently Itanium ABI [2] mandated the layout and ordering of VTables, ordering of function pointers in the VTables, sizes of VTables, etc. A conformant compiler was forced to adhere to the ABI and therefore incorporates the ABI bias that aids in reverse engineering. Even after it was changed, this bias lingers in compilers because it was fully backwards compatible. In the case of C++ programs, using the VTable as a starting point, a reverse engineer can: (1) reconstruct polymorphic classes, (2) establish polymorphic member function association by examining the function pointers in the VTable, and (3) identify inheritance relationship by either examining the construction/destruction order etc.

Bias sensitivity. Some biases are *sensitive* to change, i.e., they play a significant role in ensuring correct functioning, backward compatibility, and interoperability of binaries. For example, encrypting the virtual function pointers in VTables will cause interoperability issues. That is, classes in binaries that encrypt function pointers can not inherit from classes in binaries that do not encrypt function pointers, and vice versa.

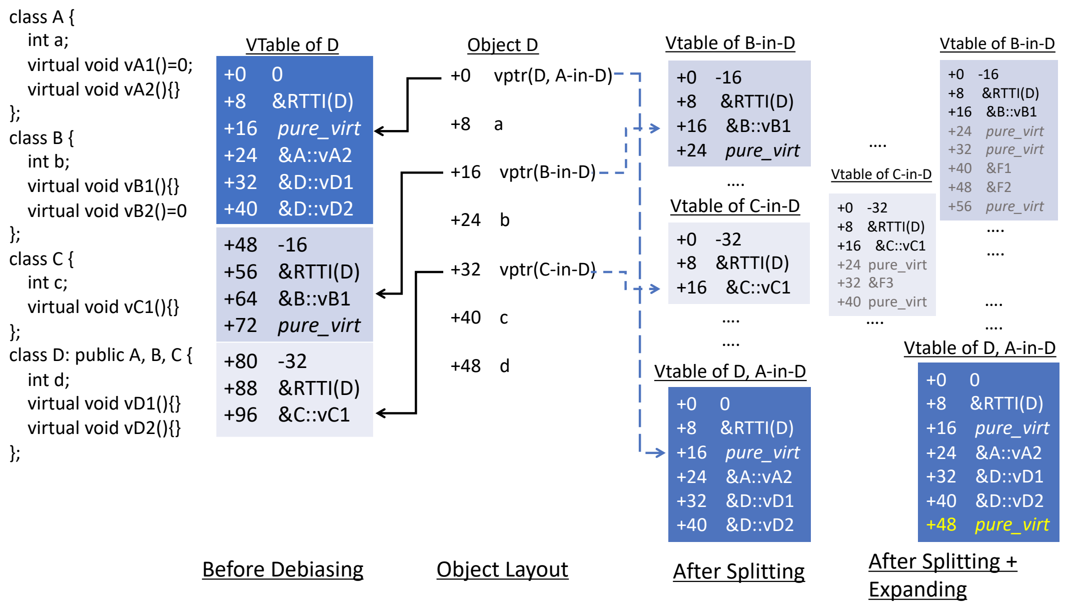
However, some biases are *insensitive* to change, and eliminating or disrupting them does not cause errors. For example, the Itanium C++ ABI used to require that the primary and secondary VTables of a complete-object VTable to be laid out one after the other in successive order of inheritance. Because the derived object and base class sub-objects contain pointers to their respective VTables, collocation of primary and secondary VTables has no benefit other than implementation convenience. *Eliminating or disrupting such insensitive biases can substantially hinder reverse engineering while ensuring no side-effects on program execution.* In this work, we identify and target two specific forms of insensitive biases – VTable bias and Function Pointer bias – in the ABI that aid in reverse engineering.

Lingering biases. The Itanium ABI [2] was modified to remove some of the requirements that lead to the Biases we identify in the following sections. This change has not been reflected in the LLVM or GCC compilers default compilation options. Instead both of these continue to follow the outdated requirements, because the new ABI specifications does not require a change, and instead simply adds a paragraph removing a guaranty [2] §2.5.2, last para. This results in the bias remaining as a security problem, even after it could be removed in an ABI compliant fashion.

Virtual Table Bias (VTB). The VTable bias is an insensitive bias that manifests due to two specific ABI mandates. First, the ABI *requires* that primary and secondary VTables be laid out in order of inheritance. That is, if class A inherits from classes B, C and D in that order, then:

$addr_{VTable(A)} = addr_{VTable(B-in-A)} < addr_{VTable(C-in-A)} < addr_{VTable(D-in-A)}$ Second, the ABI previously *required* that the primary VTable in memory be immediately followed by the secondary VTable(s) in the order of inheritance, and modern compilers stick to this in default compilation.

A reverse engineer (see vfGuard [23]) takes advantage of these requirements by (1) extracting all the VTables in the memory using a signature based approach comprising of the offset-to-top field, RTTI field and function pointers, (2) examining the offset-to-top



```
class A {
   int a;
   virtual void vA1()=0;
   virtual void vA2(){}
};
class B {
   int b;
   virtual void vB1(){}
   virtual void vB2()=0
};
class C {
   int c;
   virtual void vC1(){}
};
class D: public A, B, C {
   int d;
   virtual void vD1(){}
   virtual void vD2(){}
};
```

Figure 3: VTable layouts and contents before and after debiasing. The RTTI field is optional and contains a value 0 if the program is compiled with the -fno-rtti flag.

field to identify primary VTables wherein offset-to-top == 0, and finally (3) grouping all the subsequent secondary VTables wherein offset-to-top ≠ 0.

Note that unlike VTable bias, offset-to-top being zero or non-zero is a sensitive bias. Although offset-to-top does reveal information to a reverse engineer, its integrity is essential for correct functioning of the program, and therefore can not be changed.

Function-Pointer Bias (FPB). The function pointer bias is an insensitive bias that manifests due to the ABI requirements on how functions are laid out in the VTable. The ABI dictates the number of function pointers in the VTable. Specifically, the number of function pointer entries in a VTable is equal to the number of polymorphic functions accessible by an object of the class to which the VTable corresponds. This presents an inference point to a reverse engineer. Past efforts (e.g., [10, 11]) have relied on the size primitive to infer directionality of inheritance between classes. For example, given two VTables for classes A and B, the ABI mandate guarantees that B can only inherit from A if size of B's VTable is larger than or equal to the size of A's VTable.

3.1 Debiasing

In this work, we aim to balance the playing field by eliminating the lingering and insensitive VTB and FPB. We call this process debiasing. We are guided by the following goals.

Goals.

- *Program Obfuscation*: Our primary goal is to *debias* program binaries in order to hinder reverse engineering. Particularly, we wish to eliminate VTable and function-pointer biases that have proven crucial in C++ binary reverse engineering [12, 23, 25, 28].
- *Zero false positives*: Zero false positives is a necessary condition for practical adoption of any solution. False positives arise when debiased binaries generate previously non-existent faults. That is, our changes must ensure that the process of eliminating bias does not interfere with the intended use of the ABI or the program logic.
- *Backward Compatibility and Interoperability*: We wish for our changes to be transparent to other binaries that interact with the debiased binary. That is, binaries hardened using our techniques must seamlessly inter-operate with other binaries including binaries that are already deployed on a system as long as they adhere to the same ABI.
- *Near-Zero Performance Overhead*: Our solution does not modify the program logic. As such, any overhead imposed by our debiasing is a result of micro-architectural differences that are outside our control. In any case, we aim to achieve close-to-zero performance overhead in both binary size and program speed.

Obfuscation and Debiasing. Broadly, debiasing is an obfuscation technique that modifies a binary to make reverse engineering more challenging. It differs from traditional obfuscation techniques in two key ways. First, obfuscation primarily focuses on manipulating code and in a binary to hide program logic, whereas debiasing is not concerned with program logic, but rather the interfaces. Second, obfuscators must obey the rules of ABI. Therefore, irrespective of the amount or levels of obfuscation, ABI-centric reverse engineering will always yield inference points that are a result of ABI bias. Debiasing aims to identify the bias introduced by the ABI and systematically eliminate them. These differences make debiasing a complementary and orthogonal technique to traditional obfuscation approaches. Combining debiasing with traditional obfuscation significantly increases reverse engineering challenges.

3.2 High-Level Approach

At a high level, our approach is comprised of two disjoint, orthogonal phases. The VT Splitter phase is targeted at VT Bias and the VT Expander phase is targeted at FP bias. An example of our approach is presented in Figure 3. It contains a simple C++ inheritance structure where class D inherits from A, B and C. The object layout for D where A-in-D sub-object shares the base with D and B-in-D and C-in-D sub-objects are located at offsets base+16 and base+32 respectively. Notice how the complete object VTable for D contains primary and secondary VTables that are collocated. There is no logical basis to require collocation except for the ease of implementation.

VT Splitter. In the VT Splitter phase, a complete object VTable is first split into primary and secondary sub-VTables, and these VTables are randomly distributed across the read only sections of the binary. Because the vptrs in the object are the only legal reference to the VTables, splitting and redistributing the VTables disrupts reverse engineering. Particularly, VTable-grouping-based reverse engineering approaches (e.g., vfGuard [23]) are disrupted while ensuring correct functioning of the program including interoperability and backward compatibility.

VTable grouping through object analysis: Because vptrs in the object point to the respective secondary VTables, object analysis can be applied to perform VTable grouping. However, it presents significant challenges. The vptrs are initialized in respective constructors/destructors, and correct VTable grouping requires precise interprocedural static analysis, which can be hard. Moreover, constructors are aggressively inlined by C++ compilers. So, distinguishing between levels of inheritance can be hard. Specially in higher levels of optimization, the compiler eliminates constructor calls that are deemed trivial (e.g., default constructors). For example, the inlined constructor sequences for example in Figure 3 and a case where D inherits from A and B, and B inherits from X and C is indistinguishable. Whereas with collocated VTables, a reverse engineer can clearly demarcate complete-object VTable boundaries. Finally, some solutions [7] are known to rely on destructor analysis. But destructor analysis is only reliable when destructors are virtual, which may not be the case for all classes. Non virtual destructor analysis is plagued by the same inlining and optimization problems as constructor analysis.

VT Expander. This phase aims at disrupting FP bias where a reverse engineer can rely on the size of a VTable (i.e., number of function pointers in a VTable) in order to infer directionality of inheritance. VT Expander adds additional *unreachable* function pointer entries (i.e., dummy entries) into the VTable so as to artificially inflate the sizes of VTables. These function pointers are added based on an expansion factor f that is designed to normalize the sizes of the VTables in the binary. That is:

$$VT_{New\_Size} = (1 + f)\, VT_{Old\_Size},\ 0 \leq f \leq 1$$

In Figure 3, the expanded VTables for primary and secondary VTables of D are presented. Notice how the number of entries have expanded in each of the sub VTables. In our approach it is possible for VTable for B-in-D to have more or less number of entries than complete object VTable for B. This uncertainty in number of function pointers introduces further challenges in reverse engineering.

As a key requirement for correctness and in order to prevent introducing attack space, the function pointers added by our solution must be unreachable by user code. Yet it must be hard to statically reason as unreachable functions, otherwise a reverse engineer could simply exclude the function pointers as artificial.

As a solution, we derive insights from the fact that function pointers in a VTable are never individually referenced and are always referenced from the base of the VTable. Therefore, we ensure that all references to VTables are unaltered and adjusted to refer to the newly expanded VTable after debiasing. Furthermore, because the expansion factor f normalizes the size of the VTable across all the VTables in the binary, we are guaranteed that each function pointer offset in every inflated VTable is a valid reachable offset in some other VTable. Therefore, static binary analysis can not exclude a particular function pointer offset as unreachable. Our approach is confined to modifications to the VTables (data), so the solution is orthogonal to other compiler-based security solutions which rely on code modifications such as StackGuard [5].

4 DEBIASING VT BIAS

We debias VT bias through the process of VTable splitting, where each complete-object VTable is separated into sub-VTables and dispersed across the read-only section of the binary. Our implementation uses a multi-step LLVM Intermediate Representation (IR) level pass which is enabled/disabled at the command line.

4.1 Identifying VTables in the IR

Algorithm 1 General workflow of VTable splitting LLVM pass

```
1: procedure VTSPLITTER($modules$)
2:     for each $module$ in $modules$ do
3:         for each $global$ in $module$ do
4:             if $isVT(g) \&\& hasSecVTs(g)$ then
5:                 $subVTs \leftarrow getSubVTs(g)$
6:                 for each $vt$ in $subVTs$ do
7:                     $newVT \leftarrow createEmptyVT()$
8:                     $placeRandomlyInGlobal(newVT)$
9:                     $copyVT(newVT, vt)$
10:                    $replaceRefs(newVT, vt)$
11:                    $removeVT(vt)$
12:                end for
13:            end if
14:        end for
15:    end for
16: end procedure
```

The first step in implementation is identifying the VTables in the compilation unit, then separating the VTables that contain subVTables. This process can be preformed by relying on LLVM IR level name mangling rules. Finding a VTable requires finding a symbol matching `@_ZTV#` which is the Itanium ABI-specified name for a VTable. Next, in order to see if it has a sub-VTable, we check the first field of that entry to see if it has multiple arrays as its type as shown in appendix A. Once that is done, we check to see if it contains a secondary VTable structure. The name mangling rules we rely on are tied to the Itanium ABI, so this process will function on different versions of the compiler. Because it is possible to place our pass before symbol stripping is done, this method works even when the desired result is a stripped binary that will not contain these symbols.

4.2 Creating Replacements and Randomizing their Locations

Once all VTables are found, the next step is to create as many new entries in the IR as necessary to break apart the VTables. One for the primary VTable, whose size we know from the previous step. And then one for each secondary VTable we are breaking apart. To do this we create an IR entry of the correct size and type by copying the type and linkage information from the original entry, then assign it an arbitrary name. During the creation process of a new global entry in the IR, we use the fact it may be placed after any other global entry that already exists to place it into a random location. Because LLVM lays out global objects in an order based on where they appear in the IR, this is sufficient to move the sub-VTable in the resulting binary. Once it is done and placed into its new random location, we then copy all the information from the secondary VTable to this new entry.

4.3 Fixing References

After we construct each new VTable we must then go about fixing all the references in the constructors, VTTs (Virtual Table Table) and anywhere else from the old global entry to the new location. We do this by iterating over a list that LLVM maintains of users for the old entry. If the entry is a Get Element Pointer (GEP) statement, we examine it more closely. Because the old entry is constructed like an array of pointers, and due to the way the GEP statement is constructed, we have to check each instance to see if it is referring to the index from which we copied the pointer. If it is, we create a new GEP statement that holds the new VTable instead of the old secondary VTable, and use that to overwrite the old GEP. Then we move onto the next user of the old VTable, and repeat until there are no more users. LLVM holds a complete list of users for any given global data structure, so this approach does not miss any potential adjustments to the new value.

4.4 Eliminating Old VTables

Once we have iterated over every user once for every new VTable we have created, the old global will no longer have any users (objects or instructions that point to it) and it can be safely removed. We then repeat these steps until we can no longer find any more VTables in the IR. With this done, we must then remove the old global entry, as leaving it around would defeat obfuscation purposes completely. To do this we have LLVM run its dead global code elimination pass. With all references to the original entry removed, the dead code pass will detect and delete the old entry as an unused global code if nothing points to it anymore. Using this pass ensures that if we somehow missed any references to the VTable it will not be removed, and the program will still work correctly. However, in that case this particular entry will not be protected by the VT Splitter. Using the dead global code elimination pass additionally ensures that if LLVM ever updates how global code is removed, our pass will still be fully functional.

4.5 Applying the Pass

We created a custom build of LLVM 6.0, placed our pass in its source tree, and scheduled the pass as part of the default pass pipeline during compilation. We schedule our pass before LLVM performs its global dead code elimination pass for reasons stated in the previous section, and after all other code transformation passes that LLVM schedules during compilation. Applying the pass to a program's source code requires compiling it using our modified version of LLVM and Clang. This means there is no need to change any Makefiles or CMake files to enable the pass, to load the pass' library using LLVM opt or to give the pass' library as an argument to the Clang driver. This simplifies our testing procedure, and allows us to evaluate against a large set of varied programs.

5 DEBIASING FP BIAS

We address FP bias by eliminating the predictability regarding sizes of VTables and inheritance relationships. We achieve this through VT Expander. It operates on a list of VTables (if the VTables are split, then individual split VTables, if not, complete-object VTables), and adds a number of extra VTable entries to it that increases the size based on the configurable expansion factor f. In the case of a VTable that contains one or more sub VTables that have been split by VT Splitter, the extra entries are appended after each subVTable.

Algorithm 2 General workflow of VTable expander LLVM pass

```
1:  procedure VTExpander($modules$)
2:      $userSeed$ ←getUserSeed
3:      for each $module$ in $modules$ do
4:          for each $global$ in $module$ do
5:              if $isVT(g)\&\&isDefined(g)$ then
6:                  $newSize \leftarrow getNewSize(g, f, userSeed)$
7:                  for $newSize - size(g)$ do
8:                      $newEntry \leftarrow getRandomFunction()$
9:                      $appendEntry(g, newEntry)$
10:                     $updateUsers(g)$
11:                 end for
12:             end if
13:         end for
14:     end for
15: end procedure
```

5.1 Selecting Extra Function Pointers

In order to make the extra entries difficult to detect, they cannot simply be copies of entries that are earlier in the VTable. They also cannot be garbage values, if they don't point to the beginning of some function or thunk, then they can be discarded by any reverse engineering effort. To combat that, we select two types of new entries. The first is simply a function selected randomly from the same module the VTable is in. While this can lead to obviously incorrect choices (if it is pointing to the main function for example), on the whole this selection makes it very difficult to tell where the appended entries begin and end. The other type of entry we add is a pure virtual function. The frequency of pure virtual function added is configurable.

5.2 Cross Module VTables

In compilation the compiler will create dummy VTables for modules where they would be used, but are not declared. These are LLVM constant objects that hold a pointer to an array of `i8*` equal to the size of the finished VTable (see appendix A). In normal compilation these dummy VTables are replaced with the full VTables as long as the size and type of the dummy VTables match. If the size and types of these dummy VTables do not match, compilation fails. For us, this means we need to pass a user-defined seed into the pass, and we use this seed and a hash based on the mangled name of the VTable in order to calculate a consistent random size for a given VTable, even across modules.

5.3 Support for Virtual Inheritance

Our solution provides inherent support for virtual inheritance. Virtual inheritance is an important feature in C++ that is used in popular libraries (e.g., stream object in ostream). Due to the complexities in supporting virtual inheritance, the ABI mandates additional structures like construction VTables and VTTs. Both VT Splitter and VT Expander do not distinguish between the types of VTables, and as such inherently split and expand all VTables including construction VTables.

5.4 Integration with VT Splitter

To allow for the VT Expander to be run along side VT Splitter (which causes extra entries to be appended onto the split off subVTables), the only requirement is that the split VTables created with VT Splitter have the same internal IR format as any other VTable. We ensure this is true so we can run the VT Expander pass in exactly the same manner as the splitter pass, setting it so the Expander pass runs after the VT Splitter pass completes.

Table 2: Table showing the number of primary VTables, the number of secondary VTables, and the combined total

Programs	VTables		
	Primary VTables	Secondary VTables	Total VTables
Doxygen	962	79	1041
FireFox/wlibxul	14635	3911	18546
Xalancbmk_r	944	70	1014
parest_r	1590	222	1812
mysqld	4195	231	4426
Spidermonkey	1597	6	1603
Nodejs	3181	111	3292

6 EVALUATION

In this section, We evaluate VTable splitting and function pointer reordering on binaries using three criteria; size, correctness and performance. In addition we evaluated the impact of VTable splitting on DeClassifier, a modern reverse engineering tool and discuss the results in section 7.

6.1 Experimental setup

We ran these experiments on Intel Core i7-4790 3.60Ghz x 8 cores with 32GB of RAM on Ubuntu 16.04.7, with glibc 2.23(Ubuntu). We modify LLVM 6.0 to add the VT Splitter and VT Expander passes in-tree. For VT Expander and the combination of both passes we chose an expansion factor f of between .1 and .4, which was selected as a range large enough to prevent directional inference, but small enough not to bloat the binary needlessly. The test set is composed of 7 binaries 4, which were selected based on their range of complexities and real world application. For size comparison we elected to use Libxul.so, which is compiled as part of Firefox's build process as the Firefox binary itself is relatively small and has few subVTables. Each was compiled with VTable splitting and function expanding under -O0, -O2 and -O3 optimization levels, along with a ground truth case, and combined splitting and expanding. The ground truth is obtained by compiling each of the programs with default compiler options at their respective optimization levels. This makes a total of 12 binaries per application. Lastly, we compare the binary sizes and performance rate of the programs compiled with each technique against the ground truth. We use a subset of this set for our tests against DeClassfier. This subset consists of binaries compiled at -O0, as the goal is simply to demonstrate that VTable splitting interferes with its ability to correctly recover the completed object VTable layout, even in its most ideal use case.

Table 3: Performance tasks for each binary

Program	Execution Payload
Doxygen	Doxygen on itself. Avg of 5 runs
FireFox/wlibxul	Dromaeo recommended tests
Xalancbmk_r	runcpu
parest_r	runcpu
mysqld	Tests in the auth_sec suite
Spidermonkey	Test suite jit-test.py. Avg of 5 runs
Nodejs	12 benchmark tests

Performance testing. For performance, we measured the average change in execution time for varying tasks suited to each program, listed in table 3. We ran tasks that did not include multiple sub tasks (i.e., were not part of a test suite) five times and averaged the results.

6.2 Binary Size

In this section we show the change in binary size of programs compiled with VT Splitter only, VT Expander only, and both techniques combined. For VT Splitter 4, we record an average increase of 0.02%. This is expected since we keep the same number of overall VTable entries in the binary. The small overall size increase comes primarily from the code required to use the VTables in constructions being slightly larger, because they cannot be loaded from their base VTables address via an offset.

For VT Expander 4 we record an average increase of 1.40%. This falls within our expected results as we are increasing the overall number of VTable entries in the binary by a range between 10% and 40%. Table 2 shows the total number of VTables that VT Expander is acting on, and we expect an absolute size increase to any given binary relative to the number of VTables that are expanded. For binaries compiled with both techniques, we record an average percentage change of 1.42%, which is consistent with our other results.

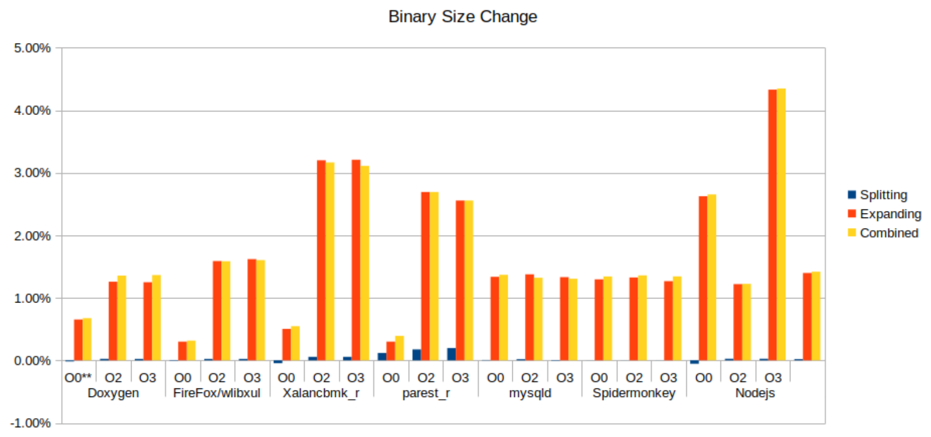


Figure 4: Binary size overheads for splitting, expanding, and a combination of the two.

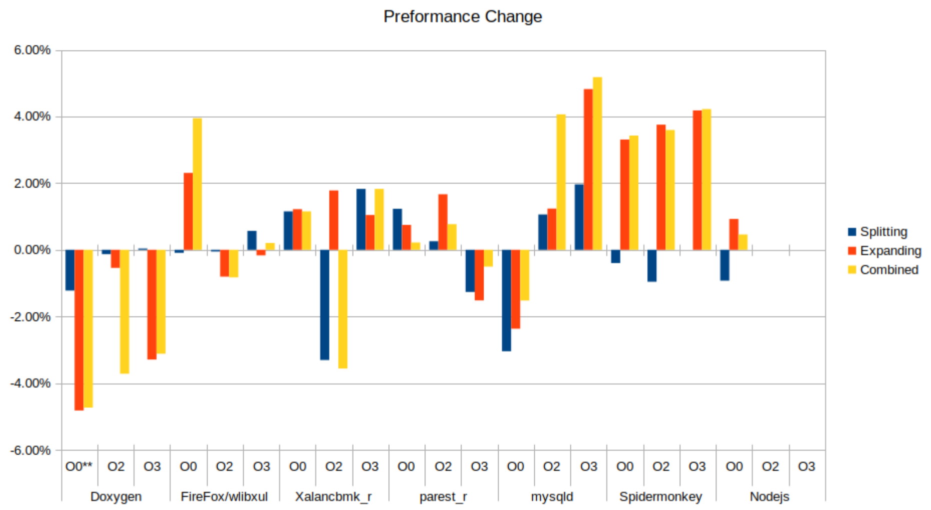


Figure 5: Performance overheads for splitting, expanding, and a combination of the two.

Negative change in binary size. Several of the -O0 results indicate a negative file size change, and in general the -O0 size increase appears quite low across almost all of the tests. This is due to the fact that our implementation takes advantage of LLVM's dead global elimination pass to safely and cleanly remove old VTables from the IR after they have been updated. Because we force this pass to run after VTable splitting or expanding it also removes any other dead globals it can detect. For the baseline -O0 this pass is not run, leaving the dead globals in the binary.

Identical baseline size. We observed an anomaly in our evaluation for spidermonkey at -O2 and -O3. Despite adding the correct optimization flags, we observe that the binary sizes for this program remain identical across two optimization levels. However despite the identical file sizes, the hash for each of these binaries is different, we can only surmise that there is some part of the compilation process which enforces this similarity.

6.3 Correctness and Performance

VT Splitter, VT Expander and both combined were evaluated for correctness and performance (figure 5). For correctness all the tests and benchmarks run passed and functioned as expected. This justifies our claim that both VTB and FPB are insensitive biases. For VT Splitter, we record an average decrease in the execution time for the selected tasks of -0.01%. This low performance overhead is expected due to the low impact that splitting the VTables apart in memory has on the code generated by the compiler. This code needs to use larger instructions to load the sub-tables in a very limited number of places. Likewise for VT Expander we observe an average penalty of 0.48%, In this case the overhead is from locality issues caused by the larger VTable sizes. Lastly, for binaries compiled with both techniques, we record an average increase of 0.66%, which is consistent with a combination of the overheads when the passes are taken separately.

Negative performance overhead. Several of the programs show negative results in overall performance overheads. Doxygen demonstrates a fairly consistent gain of around 3% under expanding and combined expanding and splitting. Or investigation of this phenomenon has shown no specific reason why this might be the case. The layout of VTables in memory does not seem to affect performance, and the number of references to the expanded VTables inside the code does not seem to be affected in any significant way (i.e., the number of times the VTable is dereferenced). No additional function inlining gets enabled by this technique, nor are there shorter code sequences for each dereference. That is, the location of the sub-VTable is still loaded, and then an offset is used as part of the call instruction. Given these findings, we can only conclude that the negative performance overhead is due to the margins of errors for most of our tests being greater than the overall change in performance, which leads to variance around the average case's relatively lower overhead.

7 SECURITY ANALYSIS

In this section, we discuss the security implications and potential ways a reverse engineer/attacker can detect or take advantage of the debiasing efforts of VT Splitter and VT Expander.

7.1 Impact on Reverse Engineering

By design, VT Splitter and VT Expander do not alter program logic, as such the reversibility of program logic is unaltered by our solution. Further, since VT-Expander inflates VTables with unreachable function pointers, its efficacy depends on the reverse engineer's ability to identify newly added function pointers. If an analyst can delineate real and inflated function pointers, they can simply exclude the inflated pointers and apply size comparisons to infer inheritance direction.

VT-Expander ensures that only reference to all function pointers in all VTables are through the base of the VTables. Therefore, any possible information leakage through code references are avoided. Second, it is possible that an analyst can detect function pointer offsets at virtual function callsites (e.g., vfGuard [23] or VTInt [28]). However, these offsets will only reveal that there exists some VTable in which the detected offset is valid. Because VT-Expander normalizes VTable sizes across all VTables in the binary, each inflated offset will also be a legal offset in some VTable in the binary. Therefore, no additional information is revealed to the analyst. Finally, by inserting *pure_virtual* functions as function pointers in the inflated VTables, VT-Expander introduces an additional dimension of uncertainty with respect to inheritance. Although not a requirement, it is common for pure virtual functions defined in base classes to be overridden by concrete implementations in the derived class. By randomly introducing *pure_virtual* function pointers in the VTable, any such inference mechanisms are hindered.

7.2 DeClassifier

We demonstrate the effectiveness of VT Splitter against DeClassifier by running it against the subset of binaries selected is shown in table 4. While VT Splitter does not prevent DeClassifier from recovering the correct number of primary VTables, it does cause it to misidentify the groupings that these VTables belong to. This prevents DeClassifier from being able to recover the correct completed object VTable for the incorrect groups as part of the scanning and grouping process described in 2.4. Without these completed object VTables you cannot match the VTable group to a single class in the source code. The incorrect groupings may also lead to an incorrect understanding of how the binary being examined functions.

Table 4: Table showing the number of groups DeClassfier found in the baseline compiler, and the total groups found after applying VT Splitter, broken up into correct and incorrect grouping

Programs	Baseline Groups	VTSplitter Groups	Incorrect Groups	Correct Groups
Doxygen	65	44	37	7
Xalancbmk	17	27	13	14
Parest	82	90	71	19

7.3 Reachability and VT Expander

By performing a reachability analysis on a binary that has been modified by VT Expander, an attacker may discount some – if not all – of the added dummy entries. Consider object D in Figure 3. A call to virtual function &B::vB1 invoked on object B-in-D would appear in the code as follows:

```
...
load   rdi , <address of B-in-D>
                  // this pointer
//setup args
load   rax , <address of VTable B-in-D>
call   qword ptr[rax + 16]
                  //VT_Offset(&B::vB1)==16
...
```

Through static analysis, an attacker can (a) extract all virtual function callsites [23], (b) identify invoking object and corresponding vtable (through overwrite analysis [21]), and (c) identify VTable offset that is accessed [23]. By excluding offsets that were referenced at callsites for a particular object type, the dummy VTable entries introduced by VT Expander can be identified.

We propose two approaches to make static analysis hard:

1) Use of pure virtual functions: We introduce *pure_virtual* entries in the VTable. These entries are not reachable (by definition) and are meant to be overridden in a derived class. One or more pure virtual function pointers that follow legitimate entries in a VTable can not be excluded as dummy entries through static analysis.

2) Dead code addition: We propose adding statically indeterminable yet dead code into the binary that comprise of virtual function invocation to the dummy function pointers. For example, calling & F1 in B-in-D's VTable would take the form of *if(expr){ d->F1(); }*. The expression *expr* is carefully chosen such that it always resolves to false at runtime, but can not be proven to be false through static analysis. Ideally, the dead code would be instrumented into the IR. To further complicate static analysis, the expression could be modeled as a function of user input and/or indirectly referenced memory.

7.4 Impact on Exploitation

Exploits in the past have leveraged function pointers in VTables as gadgets in order to execute code-reuse attacks [25]. Therefore, inflated VTables generated by VT Expander—in its current form—*may* increase the overall gadgets available to such attacks. However, the techniques incorporated by VT Expander are generic and extensible. VT Expander can be modified to emit both functions and pointers to those functions in the debiased VTables such that the functions do not perform any meaningful execution (as opposed to gadgets in code reuse attacks that must perform some useful action).

8 RELATED WORKS

Generally, binary analysis tools rely on the ability to recover semantics from the binary based on the specifications of the ABI. SmartDec [10] attempts to statically recover multiple C++ specific language constructs including classes, inheritance tree, virtual and non-virtual member functions, calls to virtual functions and exception handling. Most of the proposed techniques are dependent on the ABI specification.

VTI [3] uses the new ABI VTable rules to break apart the complete object VTables and re-arrange them in process memory in order to protect against *VTable hijacking*. While their work splits VTables apart, the reordering they perform does not remove information from the binary, only changes the layout in a *different* predictable manner.

VCI [6] recovers class hierarchy from a binary by performing constructor analysis. This analysis is based on the specification which states that the constructor of a derived class calls those of its bases. Constructor analysis simply identify such constructor calls to recover at least a partial class hierarchy tree. Similarly, Marx [21] performs overwrite analysis which is also based on the operations performed in constructors/destructors. Overwrite analysis is more robust than constructor-only analysis since it is largely unaffected by inlining which is as a result of optimization.

DeClassifier [7] is built to recover class hierarchy from optimized binaries. It combines multiple techniques to recover as much information as is present in the binary. It performs constructor/destructor analysis, overwrite analysis, and object layout analysis. Overwrite analysis allows Marx to only group related classes into sets with no direction of inheritance, while object layout analysis allows DeClassifier to assign direction of inheritance relationships identified using overwrite analysis.

ROCK [18] performs both statistical and behavioral analysis to recover class hierarchy from the binary. The behavioral analysis is based on the ABI specifications such as constructors, VTable size, position of virtual function pointers in the VTable.

vfGuard [23] proposed the VTable identification analysis which it uses to enforce CFI policy at indirect callsites in the binary. The VTable identification analysis is based on the well defined structure of the VTable which makes identifying them robust.

OOAnalyzer [24] and OBJDigger [16] recover methods and group them into classes. While OBJDigger adopts symbolic analysis and inter-procedural data flow analysis to achieve this, OOAnalyzer, a more recent tool adopts Prolog-based reasoning combined with binary and symbolic analysis. OBJDigger tracks the usage and propagation of the `this` pointer to identify related methods. Like OBJDigger, OOAnalyzer first identifies methods called using the same `this` pointer and then uses reasoning rules to decide if they belong to the same class.

TVIP [12] and VTint [28] are defenses against VTable hijacking attacks. VTint recovers VTables and appends IDs to each of them to ensure that only valid VTables are used at runtime. Like TVIP, VTint also ensures that allowable VTables point to read only section of the binary.

9 CONCLUSION AND FUTURE WORK

In this work we present the notion of *ABI Bias* which an attacker can exploit to reverse engineer a binary. Further we classify ABI biases as *sensitive* and *insensitive*. We identify two *ABI Biases* which are insensitive to change but give away crucial security information, VTable ordering bias and function pointer bias. We present the notion of *lingering bias*: bias which remains until corrected due to backwards compatibility. We present an LLVM compiler-based solution that can eliminate these insensitive and lingering biases. Our evaluation shows that the techniques introduced have little impact on the binary size and performance. Finally we argue that moving forward, the design of ABIs should take these biases and the information they can introduce into the binary into account.

The insensitive biases we discuss in depth within this paper are the ones for which we have developed an automated solution. Future work will focus on other forms of biases we feel are also insensitive, such as function pointer ordering. We also wish to investigate possible defenses against Overwrite Analysis, which is a reverse engineering technique we did not address in this paper. Finally we would like to explore the information these biases can reveal about the original source code, potentially including scoping information which is generally considered lost by the field of reverse engineering.

ACKNOWLEDGMENTS

We would like to thank our shepherd Zhiqiang Lin and the anonymous reviewers for their valuable feedback. This research was supported in part by Office of Naval Research Awards #N00014-17-1-2929 and #N6833521C0446, and DARPA award #81192. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] 2016. Itanium C++ ABI change. https://github.com/itanium-cxx-abi/cxx-abi/pull/7.
[2] 2017. Itanium C++ ABI. http://itanium-cxx-abi.github.io/cxx-abi/abi.html.
[3] Dimitar Bounov, Rami Gökhan Kıcı, and Sorin Lerner. 2016. Protecting C++ dynamic dispatch through vtable interleaving. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16).*
[4] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security.*
[5] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium (USENIX Security 98).*
[6] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. 2017. Strict Virtual Call Integrity Checking for C++ Binaries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security.*

[7] Rukayat Ayomide Erinfolami and Aravind Prakash. 2019. DeClassifier: Class-Inheritance Inference Engine for Optimized C++ Binaries. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS'19)*.

[8] Rukayat Ayomide Erinfolami and Aravind Prakash. 2020. Devil is Virtual: Reversing Virtual Inheritance in C++ Binaries. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 133–148. https://doi.org/10.1145/3372297.3417251

[9] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-based Control Flow Integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*.

[10] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *Reverse Engineering (WCRE), 2011 18th Working Conference on.*

[11] A. Fokin, K. Troshina, and A. Chernov. 2010. Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *2010 14th European Conference on Software Maintenance and Reengineering*. 240–243.

[12] Robert Gawlik and Thorsten Holz. 2014. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proceedings of 30th Annual Computer Security Applications Conference (ACSAC'14)*.

[13] CXX-ABI Discussion Group. [n.d.]. CXX-ABI-Dev mail archives. https://www.mail-archive.com/cxx-abi-dev@codesourcery.com/index.html.

[14] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. ShrinkWrap: VTable Protection without Loose Ends. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*.

[15] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of 21st Annual Network and Distributed System Security Symposium (NDSS'14)*.

[16] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. 2014. 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)*.

[17] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, Brecht Wyseur (Ed.). IEEE, 3–9. https://doi.org/10.1109/SPRO.2015.10

[18] Omer Katz, Noam Rinetzky, and Eran Yahav. 2018. Statistical Reconstruction of Class Hierarchies in Binaries. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*.

[19] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*.

[20] Nathan Burow and Derrick McKee and Scott A. Carr and Mathias Payer. 2018. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*.

[21] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX : Uncovering Class Hierarchies in C++ Programs. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium.*

[22] Pawlowski, Andre and van der Veen, Victor and Andriesse, Dennis and van der Kouwe, Erik and Holz, Thorsten and Giuffrida, Cristiano, and Bos, Herbert. 2019. VPS: Excavating High-Level C++ Constructs from Low-Level Binaries to Protect Dynamic Dispatching. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*.

[23] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*.

[24] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. 2018. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables. In *2018 ACM SIGSAC Conference on Computer and Communications Security.*

[25] Felix Shuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming, On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of 36th IEEE Symposium on Security and Privacy (Oakland'15)*.

[26] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of 23rd USENIX Security Symposium (USENIX Security'14)*.

[27] Chao Zhang, Scott A Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. 2016. VTrust: Regaining Trust on Virtual Calls. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*.

[28] Chao Zhang, Chengyu Song, Zhijie Kevin Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Defending Virtual Function Tables' Integrity. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*.

A VTABLES IN LLVM-IR

Listing 1: Virtual inheritance dimond structure in LLVM-IR broken up to show the individual sub-vtables

```
@\_ZTV1D = unnamed\_addr constant {
    [6 x i8*], [5 x i8*], [4 x i8*],
     [5 x i8*] } {

 [6 x i8*] [i8* inttoptr (i64 48 to
    i8*), i8* inttoptr (i64 32 to i8
    *), i8* inttoptr (i64 16 to i8*)
    , i8* null, i8* bitcast ({ i8*,
    i8*, i32, i32, i8*, i64, i8*,
    i64 }* @\_ZTI1D to i8*), i8*
    bitcast (void (%struct.D*)*
    @_ZN1D1fEv to i8*)],

 [5 x i8*] [i8* null, i8* inttoptr (
    i64 16 to i8*), i8* inttoptr (
    i64 -16 to i8*), i8* bitcast ({
    i8*, i8*, i32, i32, i8*, i64, i8
    *, i64 }* @\_ZTI1D to i8*), i8*
    bitcast (void (%struct.C*)*
    @_ZN1B1gEv to i8*)],

 [4 x i8*] [i8* inttoptr (i64 -32 to
    i8*), i8* inttoptr (i64 -32 to
    i8*), i8* bitcast ({ i8*, i8*,
    i32, i32, i8*, i64, i8*, i64 }*
    @\_ZTI1D to i8*), i8* bitcast (
    void (%struct.D*)*
    @_ZTv0_n24_N1D1fEv to i8*)],

 [5 x i8*] [i8* null, i8* inttoptr (
    i64 -16 to i8*), i8* inttoptr (
    i64 -48 to i8*), i8* bitcast ({
    i8*, i8*, i32, i32, i8*, i64, i8
    *, i64 }* @\_ZTI1D to i8*), i8*
    bitcast (void (%struct.C*)*
    @_ZN1C1gEv to i8*)]

 }, align 8
```

...