On Design Inference from Binaries Compiled using Modern C++ Defenses

Rukayat Ayomide Erinfolami
Binghamton University

Anh Quach
Binghamton University

Aravind Prakash
Binghamton University
{rerinfo1,aquach1,aprakash}@binghamton.edu

Abstract

Due to the use of code pointers, polymorphism in C++ has been targeted by attackers and defenders alike. Vulnerable programs that violate the runtime object type integrity have been successfully exploited. Particularly, virtual dispatch mechanism and type confusion during casting have been targeted.

As a consequence, multiple defenses have been proposed in recent years to defend against attacks that target polymorphism. Particularly, compiler-based defenses incorporate design information—specifically class-hierarchy-related information—into the binary, and enforce runtime security policies to assert type integrity.

In this paper, we perform a systematic evaluation of the side-effects and *unintended consequences* of compiler-based security. Specifically, we show that application of modern defenses makes reverse engineering and semantic recovery easy. In particular, we show that modern defenses "leak" class hierarchy information, i.e., design information, thereby deter adoption in closed-source software. We consider a comprehensive set of 10 modern C++ defenses and show that 9 out of the 10 at least partially reveal design information as an unintended consequence of the defense. We argue a necessity for design-leakage-sensitive defenses that are preferable for closed-source use.

1 Introduction

The benefits of C++ as an object-oriented language have prompted its wide use in commercial software. As a consequence, the under-the-hood mechanisms of the language implementation (e.g., virtual dispatch) have come under strict scrutiny from both the attackers and defenders. Particularly, practical attacks against C++ software that target control-flow hijacking through virtual dispatch [25, 26], and type confusion through static and dynamic casts [12, 17] have become commonplace. As such, in the last few years, the defense community has focused on defending against such attacks. Defenses both at source-code [4, 15, 16, 27] and binary levels [6, 20] have been proposed. Compiler-based defenses that rely on source code utilize rich high-level class inheritance information available in the source code and construct strict integrity (control-flow integrity in the case of virtual dispatch and type integrity in the case of type confusion attacks) policies.

On the one hand, with access to source code, compiler-based defenses are precise and well-performing when compared to binary defenses, and so, recent research in protection of C++ software has primarily leaned towards compiler-based defenses [4, 15, 16, 27]. On the other hand, the unintended consequences (side effects) of such defenses have been overlooked, receiving little to no attention. In this paper, we systematically analyze 10 C++ compiler-based defenses to examine their effect on binary reverse engineering. Our results show that 9 out of 10 defenses reveal sensitive class hierarchy information as an unintended consequence. From a software design standpoint, designing class hierarchy is pivotal to a software's success, and is therefore highly valuable.

From a security perspective, both control-flow-hijacking and type-confusion attacks originate from abuse of inheritance and polymorphism in C++. In essence, inheritance and polymorphism in C++ are defined through the classes and their relationship, i.e., the class inheritance tree. As a common characteristic, compiler-based defenses analyze the source code and extract the inheritance tree, and augment sufficient information into the binary that allows for runtime validation of—at least a subset of—the inheritance tree. Such defenses have been successful in thwarting virtual-dispatch and type confusion attacks with high precision and good performance, and have been welcomed by the community (e.g., [17]).

From a software development perspective, it is crucial to prevent reverse engineering, and challenges in binary analysis that prevent accurate reverse engineering are welcome [18]. Designing classes and their hierarchy is in the heart of C++ design. Software vendors—especially for complex software, invest huge resources in design, and take stringent measures to protect their code from plagiarism and reverse engineering.

In fact, commercial software commonly use obfuscation [18] to prevent reverse engineering. In particular, com-

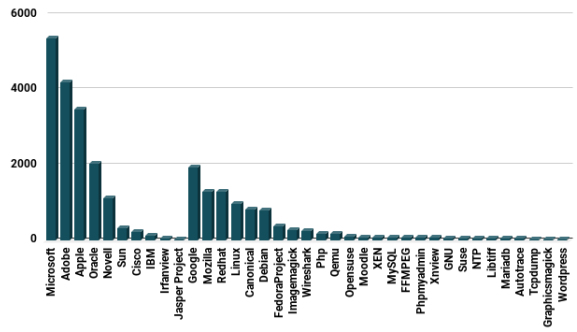

Figure 1: Vulnerabilities vs Manufacturer in the last 5 years. Source: cvedetails.com.

mercial C++ software often incorporates compile-time flag (`-fno-rtti`) to disable inclusion of inheritance-revealing runtime type information (RTTI) in the binary. RTTI includes special type-revealing data structures in the binary that allows for runtime class type resolution. In addition RTTI is known to impose undesirable runtime performance penalties [17]. Without RTTI, reverse engineering C++ binaries to recover design is considered hard and impractical for complex software [20]. This is partly due to the complexities in C++ language along with compiler optimizations that result in challenges for static and dynamic C++ program analysis. For example, intricacies arising due to dynamic dispatch of virtual functions necessitate use of indirect branches in the binary. Indirect branches pose dead-ends with respect to static analysis [28], and dynamic analysis is known to lack coverage. These challenges are a blessing-in-disguise to the closed-source vendor community.

Although modern compiler-based defenses render C++ software attack resilient, by very virtue of augmenting a binary with inheritance information, they reveal a significant aspect of design which vendors do not necessarily want to make public. Furthermore, Figure 1 shows that vulnerabilities reported in the last 5 years for closed-source software vastly outnumber the vulnerabilities reported in open-source products. Therefore, practical and impactful defenses must not only target the open-source community, but also address the concerns of the closed-source community in order to encourage adoption.

Intuitively, the addition of class inheritance information into the binary should aid in reverse engineering and design inference against modern C++ defenses, but so far, there has not been a systematic evaluation of an such effort. In this paper, we present *practical* design inference approaches against modern compiler-based C++ defenses. We show that a *significant* amount of class hierarchy information can be recovered from binaries protected by modern defenses with *high fidelity*. We considered a comprehensive list of 10 modern defenses, and based on source code availability, we evaluated 4 representative defenses. Based on our experimental findings and the technical details of the remaining 6 defenses, we believe that class hierarchy information can be successfully recovered from embedded metadata of 9 out of 10 defenses. In each case, by modeling classes as nodes and inheritance relationships as edges, we were able to recover 95% of classes with edge-correctness of over 80%, and low inheritance graph edit distance. In spite of precise security, leaking of design information is counter to the interests of closed-source community. At the very least, *closed-source developers must endure real non-negligible risk with respect to design revelation in using modern C++ defenses.* Precise and well performing defenses against control-flow-hijacking and type-confusion attacks in closed-source C++ software are needed.

Our contributions can be summarized as follows:

1. We consider a comprehensive set of 10 modern C++ defenses, and show that 9 of them at least partially reveal design (class hierarchy tree) information as shown in Table 1. We provide systematic inference strategies for such defenses.

2. We recover directed class inheritance graph for popular open-source programs and show that the recovery is of high fidelity, i.e., 95% polymorphic classes recovered with over 80% edge correctness.

3. Although we primarily target polymorphic classes, we show that both polymorphic and non-polymorphic classes can be successfully recovered from binaries compiled using 2 out of 10 defenses.

The remainder of the paper is organized as follows. Section 2 provides the technical background followed by an overview of our approach in Section 3. We present the details of our design inference approach in Section 4 followed by evaluation in Section 5. Finally we present the related work, conclusion and acknowledgement in Sections 6, Section 7 and Section 8 respectively.

2 Background

2.1 C++ Polymorphism—Under the Hood

Virtual functions are at the heart of polymorphism. In order to implement polymorphism, C++ compilers utilize a per-class supplementary data structure called a “VTable" that contains a list of polymorphic (virtual) functions an object may invoke. The structure of a VTable is dictated by the C++ Application Binary Interfaces (ABIs) – Itanium [3] and MSVC [22]. For the rest of this paper, we refer to the Itanium ABI although the differences between the two ABIs are insignificant to our inference approach. A VTable is allocated for each polymorphic class (i.e., a class that contains virtual functions, or inherits from class(es) that contain virtual function, or inherits a class virtually). Within the constructor of a polymorphic class, a

Table 1: Recoverability of Class Hierarchy Tree (CHT) from binaries compiled using various defenses. “Inference depends on callsite" means that design inference on that defense depends on callsite information, “Provides direction info" means that the class hierarchy metadata embedded by defense also includes direction of inheritance, “CHT Recovery" means how much of the embedded class hierarchy tree can be recovered"

Scheme	Category	Inference depends on callsite?	Provides direction info?	CHT Recovery
SafeDispatch [15]	Polymorphic classes-aware	✗	✓	Full
FCFI [27]	Polymorphic classes-aware	✗	✓	Full
Shrinkwrap [11]	Polymorphic classes-aware	✗	✓	Full
OVT [4]	Polymorphic classes-aware	✓	✓	Full
VIP [7]	Polymorphic classes-aware	✓	✗	Full
VTrust [29]	Polymorphic classes-aware	✓	✗	Partial
CaVer [17]	All classes-aware	✓	✓	Full
TypeSan [12]	All classes-aware	✗	✓	Full
HexType [16]	All classes-aware	✓	✓	Full
CFIXX [19]	Polymorphic classes-aware	✗	✗	Low

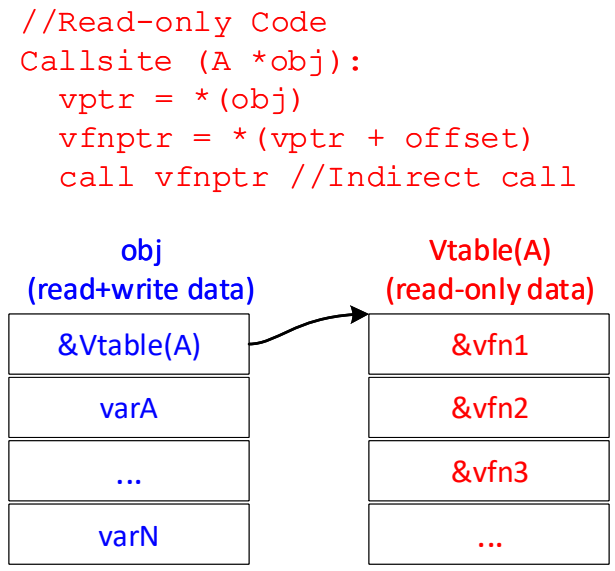

Figure 2: Virtual dispatch mechanism in C++.

pointer to the VTable – called “virtual table pointer" or vptr – is stored as an implicit member in object instance. In case of multiple inheritance, wherein a class derives from more than one polymorphic base class, the VTable for derived class comprises of a group of two or more VTables. The group comprising of the primary and secondary VTables is collectively called the complete-object VTable for the derived class. The complete-object VTable for an object is a comprehensive representation of polymorphic capabilities of an object. For more information on the structure of a VTable, we refer readers to the Itanium ABI [3].

Each virtual function and its polymorphic variants (i.e., functions that override the virtual function) are stored at a fixed predefined offset. While the offset is known at compile-time, the concrete virtual function is resolved at runtime. At a virtual function dispatch site (as shown in Figure 2), the compiler embeds code to access the VTable from the object, offsets into the VTable and retrieves the appropriate virtual function, and finally invokes the virtual function. Because the virtual functions are resolved at runtime, an indirect `call` instruction is used to accomplish the dynamic dispatch.

2.2 Attacks Against C++

The mechanisms surrounding virtual function dispatch, particularly storage and retrieval of virtual function pointers, have been targets of exploits. Based on the nature of vulnerability and the exploitation, we classify attacks into two categories.

2.2.1 VTable Hijacking

In this case, an attacker corrupts the object (or creates fake objects) such that the VTable pointer points to an illegitimate location. An example of virtual dispatch along with the sample object and the VTable layout is provided in Figure 2. The code and the VTables are usually protected by allocating them in non-writable sections. However, the objects themselves are located in writable region (heap, stack or the data region). Typically, the attacker exploits a memory corruption vulnerability to overwrite the VTable pointer within the object. Such a pointer could either point to an attacker-injected VTable (VTable injection), or an offset in an existing legitimate VTable. In both cases, virtual function pointer resolution at the virtual dispatch callsite results in a corrupted pointer. Invocation of corrupted pointer leads to arbitrary code execution.

2.2.2 Unsafe Casting

In this case, the attacker takes advantage of unsafe and inconsistent casting operations in code. C++ supports multiple types of casting operations—*static_cast, reinterpret_cast, const_cast* and *dynamic_cast*. Particularly, when an object allocated to a base class is casted to a derived class, it results in invalid downcasting. Virtual function calls using such a downcasted derived class pointer can lead to arbitrary code execution when the virtual function offset is larger than the size of the base class VTable.

2.3 Compiler-Based C++ Defenses

2.3.1 Defenses Against VTable Abuse

Defenses in this category, except CFIXX, harness the result of class hierarchy analysis (CHA) to defend against VTable abuse. They restrict the functions or vptrs allowed at a callsite to those in the specific hierarchy of the callsite's static type. While FCFI [27] builds the result of CHA into a table specifying valid vptrs for any given callsite, SafeDispatch [15] builds for both valid vptrs and valid functions. Shrinkwrap [11] modifies FCFI by removing redundant vptr entries in the metadata table, thereby providing stricter defense. This metadata is embedded into the binary to be looked up at runtime. OVT/IVT [4] attempts to reduce the overhead of vptr validation at runtime by reordering VTables in preorder traversal such that VTables within the same hierarchy are laid out contiguously. A range can therefore be assigned to specify the valid VTables in a given hierarchy. The range assigned to a type tells the set of its subclasses. OVT adds paddings in between VTables so that vptr validation can be done in a simple range check. IVT eliminates these paddings by interleaving VTable entries. More details about the specific metadata embedded by these defenses is presented in Section 4.2.

At compile time, VTrust [29] generates hash values to tag functions that are legitimate for each callsite, basically functions defined by classes in the same hierarchy. Hash values are computed using function name, argument type list, qualifiers and most base class which defined each function. At runtime, the hash value at the caller callsite is compared with that at the beginning of the callee. All polymorphic functions have the same hash values. VIP [7] supplements pointer analysis with CHA to provide a more precise set of valid vptrs. Pointer analysis makes it possible to identify types that are likely to be used at a given callsite during execution. This information helps to further restrict the set obtained from CHA. CFIXX [19] enforces Object Type Integrity (OTI) which dynamically tracks object type and enforces its integrity against arbitrary writes. OTI protects key application control flow data from being corrupted. CFIXX keeps track of the only valid object at a callsite at runtime, it does not embed metadata of class hierarchy.

2.3.2 Defenses Against Unsafe Casting

CaVer [17] uses a runtime type tracing mechanism called type hierarchy table (THTable) to dynamically verify type casting operations for polymorphic and non polymorphic classes. Given a pointer to an object allocated as type T, the THTable stores the set of all possible types to which T can be casted (i.e. its base classes) including its phantom classes. TypeSan [12] improves CaVer by providing lower runtime overhead and higher detection coverage. It uses two data structures: type layout table and type relation tables which serve the same purpose as CaVer's THTable but useful for optimizing type checks. These data structures also specify the base classes of the representing class.

HexType is an improvement over TypeSan and CaVer. It provides two folds of improvement, 1) higher coverage of typecasting operations and 2) lower performance overhead. The first is achieved by considering more instances of object creation which include the use of new operator, placement new, and when an already constructed object is hard-coded. The second is achieved by improving the method of object tracing. HexType and CaVer use the same data structures.

3 Overview

Design Inference. In this work, we propose systematic approaches to infer design information, specifically class inheritance tree from a protected C++ binary.

3.1 Compiler Defense Categories

We apply our design inference approach to a diverse and comprehensive set of C++ defenses against control-flow-hijacking and type-confusion attacks. In fact, all compiler-based defenses that result in binaries adhering to the ABI are susceptible to design inference.

Based on how design information is embedded in the binary, we group solutions into two categories.

C1: Explicit Design Information Inclusion: Solutions in this category explicitly embed design information into a given binary. Validating an object's type (e.g., object used at a given callsite) simply requires searching the information stored in the binary. In order to reverse such binaries, we extract the embedded information (typically in read-only data sections) and process them to extract class hierarchy information. FCFI [27], ShrinkWrap [11], SafeDispatch [15], Caver [17] and TypeSan [12] are examples of defenses in this category.

C2: Implicit Design Information Inclusion: These solutions transform the original design information into forms that makes verification faster or reduce the amount of work required to provide protection. For example, OVT [4] encodes class hierarchy information by choreographed ordering of existing VTables thereby eliminating the need to embed class

hierarchy information in a separate section. Similarly, VIP [7] and VTrust [29] are examples of defenses in this category.

3.2 Scope and Assumptions

Our design recovery approaches are applicable to *all* known source-code based C++ defenses against attacks that abuse polymorphism-related mechanisms in C++.

Since most C++ compiler-based defenses adhere to the Itanium ABI, our solution targets defenses that adhere to the Itanium ABI. Although, due to the high similarity between Itanium and MSVC ABIs, our technique will also apply to potential solutions on MSVC. IVT [4] is a compiler-based defense that violates the ABI. Therefore, IVT can not interoperate with binaries that adhere to the ABI. It is outside our scope.

3.3 High-Level Approach

At a high level, our approach comprises of two steps. First, given a protected C++ binary, we extract all the polymorphic classes from it. This is simply the set of all complete-object VTables in the binary. Each complete-object VTable maps to a specific polymorphic type in the program. Recall that CaVer and TypeSan represent classes, polymorphic and non-polymorphic, as THTables. Each THTable contains information such as base classes of the representing class. Second, for each type, we generate a *Type Congruency Set* or (TC_{Set}), which is simply a set of types that are congruent or exhibit a *is-a* relationship with the type. That is:

$$TC_{Set}(\mathrm{X}) = \{\mathrm{Y}, \text{where } \mathrm{Y}\ \ is-a\ \ \mathrm{X}\ \ and\ \ \mathrm{X}, \mathrm{Y} \text{ are polymorphic types } \}$$

Intuitively, $TC_{Set}(\mathrm{X})$ of a type X contains the type X itself and all the classes that derive from X because derived classes exhibit $is-a$ relationship to their bases. The TC_{Set} provides a commonality between different defenses. Defense policies for both virtual call dispatch and type casting are based on polymorphic relationship between types, which is succinctly captured by TC_{Set}.

Given the TC_{Set} for all the polymorphic types in the binary, we construct a directed inheritance graph for the binary.

Does the underlying compiler matter? Our approach relies solely on the C++ ABI that a compiler adheres to, and not on specific compiler features. Further, as a proof of concept, we evaluate defenses that adhere to Itanium ABI, which is implemented in two popular compilers GCC and LLVM. Given that the high-level design and data structures (e.g., offset-to-top, RTTI) in Itanium [3] and MSVC [22] ABIs are largely similar, we believe that with insignificant changes to our approach, defenses that adhere to the MSVC ABI can also be targeted.

3.4 Key Challenges

Unavailability of Runtime Type Information. Without access to symbol information, inferring class hierarchy is a hard but important part of C++ reverse engineering. Prior efforts have relied on the RunTime Type Information (RTTI) to recover inheritance structure. However, RTTI is an optional data structure that can be omitted if `typeinfo` and `dynamic_cast` are not used in the code. In fact, due to the rich semantic information contained within RTTI, along with performance penalties incurred in using `dynamic_cast`, commercial closed-source software often exclude RTTI information and hinder reverse engineering.

In the absence of RTTI, past efforts have relied on the sizes and contents of VTables for reverse engineering class hierarchy. For example, Fokin et al., [8, 9] derive rules between two polymorphic classes, and apply them to infer inheritance. These approaches either (1) lack in precision [1, 5], or (2) lack in directionality of class inheritance graph [20].

From a software vendor's point of view, such inaccuracies are preferable, and hinder reversing of software design. Yet, from an end-user point of view, such inaccuracies result in coarse-grained CFI policies that introduce an attack surface in the form of redundant edges in the CFG.

Compiler Optimizations. Function inlining and removal of unused code become aggressive with higher levels of optimization. These occurrences limit the amount of information available for reverse engineering. For instance, constructors give indications about the base classes of a class as well as the direction of inheritance. However, the compiler could either inline them or even remove them completely, thereby making their identification difficult and incomplete. This eventually results in incomplete or over approximation of class relations.

4 Design Inference

4.1 VTable Accumulation and Grouping

As a first step, we extract all the complete-object VTables in the binary. Complete-object VTables provide an unlabeled representation of all the polymorphic classes in the binary. In essence, they represent the nodes in the class inheritance tree of the program.

Extracting VTables from a binary is a previously studied problem [10, 21, 30]. The well defined nature of VTables, particularly the existence of mandatory fields [3], provides a robust signature that can retrieve all the VTables in the binary. Objects are initialized with a vptr in the constructors and destructors of polymorphic classes. These VTable addresses appear as immediate values. We scan the binary for immediate values that point to readonly sections of memory. We then examine each of those addresses for VTables, and accumulate all the VTables. Our approach to VTable recovery

is similar to the one presented in vfGuard [21]. However, the recovered VTables (using vfGuard approach) include both the primary and secondary VTables. This means that one or more VTables in the gathered list map to a single polymorphic class in the program. Therefore, we merge the primary and all corresponding secondary VTables to obtain a set comprising of only the complete-object VTables. We implement Algorithm 1 to group recovered VTables into complete-object VTables.

Algorithm 1 is based on two key observations:

- Modern compilers (g++ and LLVM-clang) layout primary and secondary VTables for a derived class in sequential order.
- Since offset-to-top represents the displacement from the top of the object to a sub-object, a value of 0 represents primary VTable, and a non-zero value represents secondary VTable.

Given a set of VTables, we first sort the VTables in increasing order of vptrs. Then, we merge the primary VTables with succeeding (zero or more) secondary VTables to form the complete object VTables. VTable grouping for OVT is done differently. This is necessary because OVT reorders VTable which separates secondary VTables from their corresponding primary VTables. Each VTable is placed in a tree where the most base class it inherits from is the root. We adopt two techniques to obtain the complete-object VTables for OVT binaries: thunk and constructor analysis, they are discussed in Section 4.4.

Algorithm 1 GroupVTables groups VTables in $\mathcal{V}$ to form complete-object VTables $\mathcal{V}_{Group}$ where each VTable in $\mathcal{V}_{Group}$ is a Primary VTable, and contains a list of zero or more Secondary VTables.

```
1:  procedure GROUPVTABLES(V)
2:      V_Sorted ←
3:      sort_increasing(V)
4:      V_Group ← ∅
5:      for each V_T in V_Sorted do
6:          if V_T.OffsetToTop == 0 then
7:              V_Primary ← V_T
8:              V_Group.append(V_Primary)
9:          else
10:             V_Primary.Secondaries.append(V_T)
11:         end if
12:     end for
          return V_Group
13: end procedure
```

4.2 Generating Type Congruency Set

The format in which design information is represented in the binary determines how the TC_{Set} can be recovered. FCFI, Shrinkwrap and SafeDispatch use similar representation, while CaVer and TypeSan use a different representation. TC_{Set} for FCFI, Shrinkwrap and OVT comprises of complete-object VTables while that for CaVer and TypeSan comprises of THTables.

4.2.1 C1 Defenses

Algorithm 2 shows how metadata is recovered for C1 defenses.

Algorithm 2 GatherTCSet for C1 defenses

```
1:  procedure GATHERTCSET(sAddr, eAddr)
2:      MD_Tables ← ∅
3:      addr ← sAddr
4:      while addr ≤ eAddr do
5:          addr ← get_next_8bytes_aligned_addr(sAddr)
6:          if is_valid_metadata_start(addr) then
7:              MD_Table ← ∅
8:              while !is_valid_metadata_end(addr) do
9:                  key ← extract_next_key(addr)
10:                 MD_Table.append(key)
11:                 addr ← get_next_8bytes_aligned_addr(addr)
12:             end while
13:         end if
14:         MD_Tables.append(MD_Table)
15:     end while
          return MD_Tables
16: end procedure
```

FCFI and Shrinkwrap. The VTable maps embedded by FCFI and Shrinkwrap are simply arrays of pointers to VTable with nothing indicating its start or end. However, the VTable maps are initialized with calls to a function named __VLTRegisterSetDebug, which contains details about the VTable map addresses and their lengths. As shown in listing 1, the first argument to this function is a pointer to the VTable map for a given class while the second argument is the length of the map. Once the first and second arguments to this function are identified, we can recover the complete inheritance graph for the program. After identification of these callsites and extraction of VTable map addresses and length, we locate all the VTable maps and use the length to decide the end of each map. Since Shrinkwrap puts primary and secondary VTables in different VTable maps, we use VTable grouping to locate the primary VTable of any secondary VTable we find in a map.

Listing 1: Function used to initialize VTable maps

```
...
1 init &vtable_map
2 init length_of_vtable_map
...
3 call __VLTRegisterSetDebug
...
```

CaVer and TypeSan. The data structures embeded by these defenses are well defined, starting and ending with easily distinguishable entries and stored in the data section. CaVer and TypeSan represent matadata differently but the idea is

the same, hence, we generally refer to metadata representation in both as THTables. We first perform a linear sweep through the data section to gather THTables. For CaVer, the first field in a THTable is the length of allowable casts for the class which it represents. The next field is the key for the representing class. Next set of fields are keys to the THTables of the base classes or phantom class. The final field is the name of the representing class. TypeSan divides the operation performed with THTable into two phases, using two different data structures. Even though they are laid out differently, they both specify the inheritance relationship among classes. In some cases, each structure may contain partial information for a given class, by combining them we get the complete information.

4.2.2 C2 Defenses

OVT. OVT performs both range and alignment check in a single branch. To obtain TC_{Set} for binaries compiled using OVT, we identify callsites to extract vptrs corresponding to specific types as well as the range of its subclasses. Listing 2 shows OVT's instrumentation at virtual function callsites. As explained in Section 2, each type has an associated range which specifies the number of its subclasses. OVT computes the position of the runtime type of an object within a subtree (hierarchy) by first subtracting its vptr "$vptr" from that of its static type "$a" and then performing a right bit rotation using a literal l. It then compares the resulting value with the maximum range for the object's static type, which should be greater. ($b - $a) $\gg l$ equates to the maximum range in a given subtree (class hierarchy), where $b is the last vptr in that subtree. If the static type of the object is a leaf class, i.e. has no subclass, its vptr is compared with the vptr of the type used at runtime for equality. That vptr is used if the equality check is successful, otherwise a violation is reported.

We retrieve the range for each class by following the sequence of operations performed by OVT. The range as well as the vptr are used to recover the $TC_{Set}(X)$. Specifically, we group all the VTables within the identified range into $TC_{Set}(X)$ for the callsite's static type. For every secondary VTable found in a given range, we obtain the corresponding primary VTable using thunk or constructor analysis. Algorithm 3 shows how we identify and extract the vptrs and ranges of pointer types.

VTrust and CFIXX. VTrust like OVT implicitly embeds class hierarchy into the binary by assigning similar hash values to functions defined by classes in the same hierarchy. The information provided is sufficient to group related classes into sets (similar to Marx), but does not provide direction of inheritance. CFIXX keeps track of the single correct object type that can be used for a virtual call, it neither provides information about classes in the same hierarchy nor direction of inheritance. Therefore, VTrust reveals only partial CHT in comparison with binaries without defense, whereas CFIXX reveals no more information than binaries without defense.

Algorithm 3 GatherTCSet for OVT

```
1:  procedure GATHERTCSET(tsAddr, teAddr)
2:      addr ← tsAddr
3:      while addr ≥ teAddr do
4:          I_dec ← dec_instr(addr)
5:          if !(isDerefObjectVptrInst(I_dec)) then
6:              continue
7:          end if
8:          addr ← get_next_addr(addr)
9:          I_dec ← dec_instr(addr)
10:         if !(isMoveTypeVptrInstr(I_dec)) then
11:             continue
12:         end if
13:         vptr ← I_dec.opr[2]
14:         addr ← get_next_addr(addr)
15:         I_dec ← dec_instr(addr)
16:         if isOVTCompInstr(I_dec) then
17:             VT_Tree.append(vptr, 1)
18:             continue
19:         end if
20:         addr ← get_next_addr(addr)
21:         I_dec ← dec_instr(addr)
22:         if !(isOVTDiffInstr(I_dec)) then
23:             continue
24:         end if
25:         addr ← get_next_addr(addr)
26:         I_dec ← dec_instr(addr)
27:         if !(isOVTDiffRInstr(I_dec)) then
28:             continue
29:         end if
30:         addr ← get_next_addr(addr)
31:         I_dec ← dec_instr(addr)
32:         if !(isOVTCompInstr(I_dec)) then
33:             continue
34:         end if
35:         VT_Tree.append(vptr, I_dec.opr[2])
36:     end while
          return VT_Tree
37: end procedure
```

4.3 Inferring Inheritance

We combine all recovered type congruency set into the class hierarchy for the entire binary. TC_{Sets} having similar entries are merged into a single tree, which still maintains the edges and the direction of inheritance.

4.4 Thunk and Constructor Analysis

Thunk and constructor analysis are done to identify the corresponding primary VTable of a secondary VTable found in subtrees of binaries compiled using OVT. This is necessary because OVT reorders all sub VTables of a given class into the various subtrees they belong to, which makes VTable grouping (explained in subsection 4.1) not applicable.

Every function in a secondary base class that is redefined by the derived class has a thunk entry in the Base-in-Derived

Listing 2: Callsite generated by OVT

```
...
// $a = vptr of most base class in subtree
1 $diff = $vptr - $a
2 $diffR = rotr $diff, 1
// ($b - $a) >> 1 equates to max range for $a
3 cmp $diffR, ($b - $a) >> 1
...
```

VTable. Hence, we link every secondary VTable which contains a thunk entry to the primary VTable containing the function it references. This method is certain to identify the correct primary of a given secondary VTable, however, a thunk might not always be present. For this reason, we also employ grouping VTables through constructor analysis.

One of the operations performed within a constructor is to write the vptrs (primary and secondary) of the class whose object is to be constructed in the memory space allocated for that object. To group VTables using constructor analysis, we first scan for constructors in the .text section of the binary. Next, we extract all valid vptrs from each constructor. Finally, we identify the primary VTable by looking at the offset-to-top entry which must be zero.

4.5 Executables vs Shared Libraries

Design information recoverable from shared libraries tend to be more comprehensive and accurate compared to executables. The scope of use of shared libraries is unknown at compile time. This prevents compilers from inlining their functions and makes it necessary to retain all useful information. However, executables get inlined aggressively since the scope of their use is visible to the compiler during compilation. For instance, CaVer and TypeSan embed THTables while FCFI embeds VTable maps for virtually all classes in a library.

4.6 Callsite Analysis

Callsite analysis is necessary for C2 defenses which implicitly embed class hierarchy information at callsites. The completeness of our analysis for OVT, for example, depends greatly on the number of base classes which have at least one corresponding callsite. The availability of this information at callsites makes it possible for us to identify subtrees which are merged to build the class hierarchy for the entire binary.

4.7 Handling Template Classes

Binaries generated by both GCC and Clang contain separate VTables for each template implementation in the binary, so is the ground truth obtained using GCC's -fdump-class-hierarchy. All the compiler-based defenses targeted in this work modify either GCC or Clang, therefore they also treat template classes similarly. Since our ground truth matches the binaries being analyzed, the presence of template classes does not affect the precision recorded.

4.8 Graph Similarity Measure

To show the effectiveness of our approach, we need to quantify the similarity of the class inheritance recovered from hardened binaries with the ground truth. This problem can be reduced to a graph matching problem. Classes in the hierarchy are represented as nodes, while relationships among them are represented as edges. GEDEVO [14] is an evolutionary algorithm which uses Graph Edit Distance (GED) as optimization model for finding the best similarity between two graphs. GEDEVO produces high quality result even on large graphs, this makes it appropriate for this work since our evaluation set includes large programs like Spidermonkey. GED is a general model for solving graph matching problem, it is defined as the minimal amount of modification needed a in graph G_1 to make it isomorphic to graph G_2 [2, 14].

For our evaluation, we modeled class inheritance as a directed graph containing pairs of (baseClass, derivedClass) interaction networks. Considering two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and a one-to-one mapping f between nodes V_1 and V_2. The GED between G_1 and G_2 induced by mapping f is as follows:

$$GED_f(G1, G2) = | \{(u,v) \in E_1 : (f(u), f(v)) \notin E_2\} \bigcup \{(u', v') \in E_2 : (f'(u'), f'(v')) \notin E_1\} |$$

By definition, $GED_f(G_1, G_2)$ counts inserted or deleted edges induced by the mapping f, which can be easily extended to depict node/edge dissimilarities. The raw GED for a mapping is calculated from the number of removed and added edges required to transform one graph into the other graph. The raw counts are summed up and normalized, producing the actual GED score.

Even though GEDEVO is dedicated to biological network, it internally utilizes GED for optimization which makes it applicable to general graph comparison problems outside computational biology.

Result Interpretation. GED score ranges from 0 to 1, a value close to 0 shows high similarity while a value close to 1 shows high dissimilarity. GEDEVO also computes the Edge Correctness (EC) of found edges in the two graphs. An EC value of 100% is the highest and it is possible when two graphs are either isomorphic or sub-isomorphic. GED score and Edge correctness represent the precision of recovery and accuracy of inheritance direction respectively.

5 Evaluation

We evaluate our solution on both executables and libraries. Our test suite include SPEC CPU 2006 C++ benchmarks, Spidermonkey, and four shared libraries — CplusplusThread, Attic-c-hdfs-client, LibEbml, and LibMatroska. Our choice

of SPEC CPU programs is based on the programs evaluated by the C++ defenses.

Spidermonkey and the libraries are chosen to show the impact of our solution on real world complex applications. For binaries compiled with FCFI and OVT we consider only polymorphic classes while for CaVer and TypeSan we consider both polymorphic and non polymorphic classes. In our evaluation, we answer the following questions:

1. How accurate and precise are the class hierarchies extracted from binaries hardened with compiler-based defenses?
2. How much design information do compiler-based defenses embed within binaries?
3. Can we recover a more complete class hierarchy from binaries hardened with modern C++ defenses, than state of the art binary analysis tools?

All evaluation was performed on a system with Intel Core i7-4790 CPU @ 3.60GHz x 8 and 32GiB of Memory, running Ubuntu 14.04 LTS with Linux Kernel 4.10.0. All binaries where compiled using GCC under O0 optimization.

Defenses Evaluated. FCFI, Shrinkwrap and SafeDispatch use similar techniques, therefore, we chose FCFI to represent the group. Also, Hextype is only an improvement over CaVer, therefore, we chose CaVer for this evaluation. We were unable to evaluate VTrust and VIP because of the unavailability of their source code.

5.1 Recovery for Polymorphic Classes

Table 2 shows the recovery rate, GED score and Edge correctness of polymorphic classes compared with the ground truth. We constructed the ground truth by using -fdump-class-hierarchy option of g++ which dumps VTables, their layout and inheritance relationship during compilation.

On the average, we obtained a GED score of 0.08, 0.21, 0.09 and 0.35 and Edge correctness of 98.26%, 89.97%, 96.31% and 81.14% for FCFI, OVT, TypeSan and CaVer respectively. The ground truth graph for Namd has four polymorphic classes with only two edges. Classes `PairCompute` and `SelfCompute` have `ComputeNonbondedUtil` as their base class. We were unable to recover `ComputeNonbondedUtil` from the Namd binary compiled with OVT this is the reason its GED score is 1 and its Edge correctness 0%. Also, we could not compile Spidermonkey with TypeSan and DealII, Libebml and Libmatroska with OVT. The recovery from binaries compiled with CaVer are not as precise and accurate as the other defenses, this is because CaVer has been shown to have low coverage [12] [16]. Not all objects are protected, hence fewer design information is embeded in the binary. Also, the GED score for Spidermonkey compiled with FCFI is not as close to zero as other programs, this is because FCFI omits design information for abstract classes.

5.2 Recovery for Both Polymorphic and Non-polymorphic Classes

CaVer and TypeSan protect both polymorphic and non-polymorphic classes, hence we evaluate the precision and accuracy of our recovery for all classes in the programs compiled with these defenses. TypeSan represents class names with hash values in the binary, while CaVer uses actual names. To construct the ground truth for TypeSan, we modified its source code to output the mapping (className, hashValue) during compilation. The results tabulated in Table 3 show the number of THTables recovered from TypeSan and CaVer.

The column "# Keys found" represents the total number of keys found in all THTables, however, some keys do not have a specific THTable in the binary, they are only present within other THTables. For TypeSan, all keys found have a corresponding THTable, but that is not the case for CaVer. CaVer protects only a subset of objects created at runtime, as a result, it does not dump metadata for those unprotected objects in the binary.

5.3 Class Hierarchy Tree Recoverable from the Amount of Information Embedded

Table 4 shows the number of unique classes whose metadata is present in the binary. The aim of this evaluation is to show that even though some defenses do not embed a corresponding metadata for every class in the binary, the ones present are enough to build an accurate class hierarchy. Unlike FCFI and TypeSan, CaVer and OVT embed metadata for classes depending on their use in the program. CaVer embeds a THTable for a class only if at least one instance of that class is created. Since we rely on callsite for OVT, the metadata of a class is made available only if an indirect call is made using an object of that class. However, the absence of metadata for certain classes will have no impact on our recovery as long as their keys (for CaVer) or VTables (for OVT) are found in the binary. For binaries compiled with CaVer, the THTables of classes with no base class are not necessary to build class hierarchy since they contain just one key and name. They do not give any information about inheritance. For binaries compiled with OVT, callsites for classes with no derived class are also not necessary since their range is 1. We refer to such classes as leaf nodes in Table 4. Only the non-leaf nodes are important for reconstructing class hierarchy. On the average we found callsites for 66% of non-leaf nodes, except for Namd whose only non-leaf node has no corresponding callsite. Note that we only consider polymorphic classes for OVT.

5.4 Comparison Against Marx

Marx [20] is a state-of-the-art reverse engineering tool for class hierarchy recovery which infers relationship among classes by heuristics. VTables are taken as classes since they

Table 2: Evaluation result for precision and accuracy of class hierarchy recovered from FCFI, OVT, TypeSan(TS) and CaVer(CV). GT is the Ground Truth obtained by using GCC's -fdump-class-hierarchy option. VTS&G is VTable Scanning + Grouping, i.e the total number of VTables recovered simply by using VTable Scanning and Grouping without relying on information embedded by the defenses. OVT did not compile DealII.

		Total Recovery					**GED Score**				**Edge Correctness**			
Programs	**GT(polymorphic)**	FCFI	OVT	TS	CV	**VTS&G**	FCFI	OVT	TS	CV	FCFI	OVT	TS	CV
Spidermonkey	807	795	780	-	521	805	0.33	0.35	-	0.38	88.34	76.98	-	68.80
Xalanc	975	958	673	913	531	857	0.02	0.23	0.15	0.43	100	94.66	86.76	68.94
Soplex	29	29	29	29	22	30	0.05	0.07	0.13	0.18	100	95.24	95.45	94.12
Povray	32	28	26	28	14	30	0.1	0.33	0.11	0.45	100	91.67	100	100
Omnetpp	112	110	105	111	110	107	0	0.21	0.1	0.32	100	81.25	97.06	70.59
dealII	874	717	–	687	98	746	0.12	–	0.16	0.86	97.70	–	94.91	46.70
Namd	4	4	0	4	4	3	0	1	0	0	100	0	100	100
CplusplusThread	11	11	9	11	8	11	0	0.09	0	0.2	100	100	100	100

Table 3: Evaluation result for the number of THTables and unique classes extracted from TypeSan(TS) and CaVer(CV) binaries for both polymorphic and non polymorphic classes. GT is the Ground truth obtained from the (hash, name) mapping which TypeSan and CaVer generate during compilation. *Keys found* is the number of unique keys found through the THTables. *THTables recovered* is the number of THTables found. TypeSan did not compile Attic-c-hdfs-client. FN and FP represent false negatives and positives in THTable recovery respectively.

	GT		**Keys found**		**THTables recovered**		**FP Keys**		**FN Keys**	
Programs	TS	CV	TS	CV	TS	CV	TS	CV	TS	CV
Xalanc	3075	1591	3162	1562	3162	1013	126	0	39	29
Deal	2332	448	2366	437	2366	263	124	0	90	11
Omnetpp	244	178	236	171	236	150	15	0	23	7
Soplex	115	54	123	50	123	37	12	0	4	4
Povray	249	33	211	32	211	27	14	0	52	1
Namd	21	10	29	10	29	9	8	0	0	0
CplusplusThread	26	14	23	14	23	7	1	0	3	0
Attic-c-hdfs-client	–	651	–	544	–	439	–	0	–	107
LibEbml	92	38	85	35	85	29	1	0	7	3
LibMatroska	286	293	303	288	303	262	18	0	1	5

are the only artifact left in the binary that can uniquely identify classes. Marx performs overwrite analysis which is based on the heuristic that in a constructor, the vptr of a base class gets overwritten by that of its derived class and vice versa in a destructor. Therefore, two vptrs that overwrite each other are said to be related. It also checks VTable function entries to take advantage of the possibility that a derived class may not redefine every virtual function it inherits from its base class. Therefore, if two VTables contain pointers to the same function(s) at the same offset, they are said to be related. In order to improve coverage, Marx also preforms inter-procedural data flow analysis, which uses forward edge analysis to resolve indirect control flow (to improve coverage for overwrite analysis) and backward edge analysis to take return values into account. Finally, it performs inter-modular data flow, which considers hierarchy from libraries to obtain a more comprehensive result. Even though these heuristics give strong indication about class relations, Marx only reconstructs class hierarchy as a plain set, direction of inheritance is not inferred. According to the authors, class relations recovery is a hard problem in itself and information regarding the direction of the relation is not available in binaries [20].

We analyzed xalancbmk with Marx, Figure 3 shows its representation of a subset of the inheritance graph generated. Marx was able to correctly identify related vptrs, but they are only grouped into a set. In Figure 4, we show a mapping between the ground truth and our analysis. The figure shows that not only are we able to infer class relations, we are also able to infer the direction of inheritance correctly. In the figure, base classes are above the derived classes and the broken lines show nodes in the ground truth corresponding to nodes in our analysis while the solid lines show relationship between classes.

Table 4: Evaluating the amount of class hierarchy information OVT reveals at callsites. Column with "# types with associated callsites" contains the number of classes that has at least one corresponding callsite, "#overall types recovered + VTable scanning" contains number of classes recovered from the binary using both callsite information and VTable scanning, "#leaf node" contains number of classes with no base class, and "#non-leaf node" contains number of classes with at least one base class.

Program	From Analysis				GT	
	# types with associated callsites	#overall types recovered + VTable scanning	#leaf node	#non-leaf node	#leaf node	#non-leaf node
Spidermonkey	91	768	19	72	636	171
Xalanc	427	847	275	152	668	307
Soplex	23	30	12	11	17	12
Povray	14	30	6	8	20	12
Omnetpp	45	104	27	18	85	27
Namd	0	3	0	0	3	1
CplusplusThread	4	11	0	4	6	5

Figure 3: Marx's representation of a subset of the inheritance graph generated from Xalanc

6 Related Work

6.1 C++ Attacks and Defenses

Due to the use of computed code pointers in dynamic dispatch, multiple attacks have targeted C++ programs including the recent COOP [26] attack that reuses existing virtual functions in order to accomplish malicious computation. With insufficient semantics in the binary, binary-level solutions – VTint [30], TVIP [10], vfGuard [21], RECALL [5] introduce imprecision in the defense. However, source-code based solutions (i.e., [4, 11, 15, 17, 27, 29]) must embed inheritance information into the binary in order to improve precision and offer interoperability, and therefore reveal design information.

6.2 C++ Reverse Engineering

VCI [6] reconstructs the class hierarchy of a program using constructor only analysis. Constructors give useful information about inheritance as well as its direction [23], however, due to constructor inlining, there are inadequate constructors in the binary to carry out a comprehensive static analysis. The evaluation reported for this work shows a high number of false positive and false negative inferred inheritance relationships. Marx [20] uses heuristics to reconstruct class hierarchy, such as, overwrite analysis, similar VTable entries and return values. For comprehensive class hierarchy, it also analyzes libraries used by executables. Even though Marx can infer relationship with high precision, it ignores the direction of inheritance.

Fokin et al. [9] depends on rules including analyzing VTable sizes, checking for pure virtual functions, checking for parameters of virtual functions and checking if the vptr of a class is overwritten by that of another class. However they are also not able to give precise class hierarchy.

OOAnalyzer [24] combines traditional binary analysis, symbolic analysis and Prolog-based reasoning to group methods into classes(for both polymorphic and non-polymorphic classes). Methods called on the same pointers are identified and then reasoning rules are applied to decide if they belong to the same classes. The authors mentioned that inheritance can be assigned using class size and VTable sizes. However, no evaluation was done regarding this , therefore, we cannot confirm if OOAnalyzer can indeed decide inheritance.

TVIP [10], ensures that a VTable pointer points to the read only section before it is used, leveraging the fact that VTables are always located in read only memory. Class hierarchy was not used. Similarly, VTint [30] is developed on the basis that all legitimate VTables are stored only in the read only memory. They identified the possibility of an attacker reusing existing VTables which are in read only memory. They suggested that this can be handled by leveraging class hierarchy information. TypeArmor [28] determines an approximate number of arguments prepared at callsite and the number of arguments that a function expects. This information is used to restrict the functions that any given callsite can target.

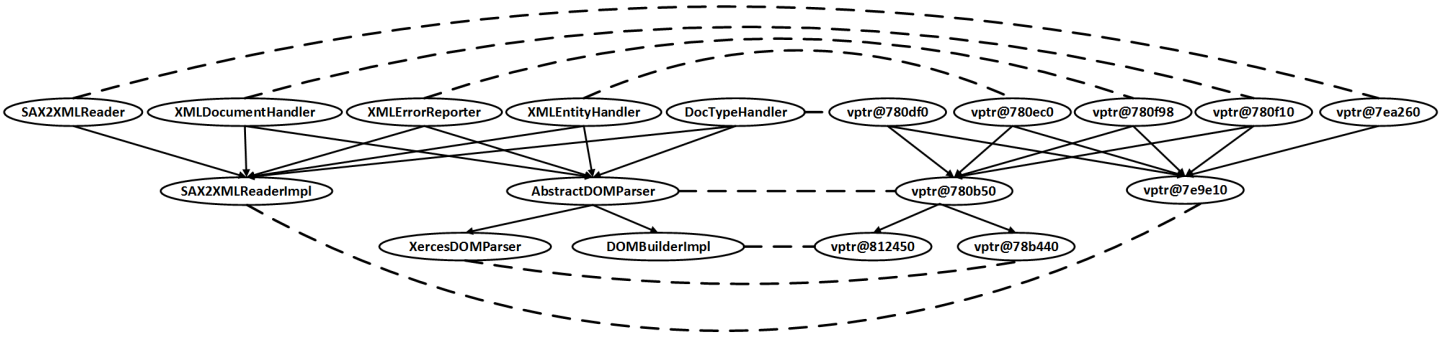

Figure 4: A mapping between a subset(for the reason of space) of the recovered class hierarchy from the ground truth and our analysis of Xalanc compiled with OVT.

7 Conclusion

In Summary, we have shown that most modern C++ defenses 1. embed metadata in the binary 2. and the metadata can be easily recovered. This makes reverse engineering the binary easier. In order to address this problem, we suggest the following:

1. Transform the metadata: This can be done either by encrypting the metadata or randomizing their layout so that recovering them does not reveal any meaningful information.

2. Avoid embedding metadata: CFIXX [19] and μCFI [13] achieve this by dynamically deciding the only valid target of a callsite. Instead of statically identifying possible targets of a callsite, CFIXX dynamically tracks object type using a policy called Object Type Integrity. This makes it possible to identify the single allowable target, thereby providing better integrity. Similarly, μCFI proposes another CFI called Unique Code Target which relies on Intel Processor Trace to dynamically record data. This data is used to augment points-to analysis which is used to identify the only allowable target of a callsite.

By incorporating design-revealing information in the binary, modern C++ defenses arguably pose a deterrent to usability in commercial products. While their precision in security is certainly appealing to the open-source software, it comes at the cost of privacy, which may not be acceptable in commercial software.

8 ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their valuable feedback. This research was supported in part by Office of Naval Research Grant #N00014-17-1-2929, National Science Foundation Award #1566532, 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] "The IDA Pro Disassembler and Debugger," https://www.hex-rays.com/products/ida/.

[2] *A Novel Software Toolkit for Graph Edit Distance Computation*, 2013.

[3] "Itanium C++ ABI," http://refspecs.linuxbase.org/cxxabi-1.83.html, Revision: 1.83.

[4] D. Bounov, R. G. Kıcı, and S. Lerner, "Protecting C++ dynamic dispatch through vtable interleaving," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[5] 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.

[6] M. Elsabagh, D. Fleck, and A. Stavrou, "Strict Virtual Call Integrity Checking for C++ Binaries," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS'17)*, 2017.

[7] S. Fan, L. Xiaokang, X. Yulei, and J. Xiangke, "Boosting the Precision of Virtual Call Integrity Protection with Partial Pointer Analysis for C++," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)*, 2017.

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

[9] A. Fokin, K. Troshina, and A. Chernov, "Reconstruction of class hierarchies for decompilation of C++ programs," in *14th European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[10] 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)*, 2014.

[11] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos, "ShrinkWrap: VTable Protection without Loose Ends," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, 2015.

[12] I. Haller, Y. Jeon, P. Hui, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "TypeSan: Practical Type Confusion Detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[13] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[14] R. Ibragimov, M. Malek, J. Guo, and J. Baumbach, "GEDEVO: An Evolutionary Graph Edit Distance Algorithm for Biological Network Alignment," in *German Conference on Bioinformatics 2013*, 2013.

[15] 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.

[16] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "HexType: Efficient Detection of Type Confusion Errors for C++," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[17] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[18] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03)*, 2003.

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

[20] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, "MARX : Uncovering Class Hierarchies in C++ Programs," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[21] A. Prakash, X. Hu, and H. Yin, "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)*, 2015.

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

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

[24] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, " Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[25] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS'07)*, 2007.

[26] F. Shuster, T. Tendyck, C. Liebchen, L. Davi, A.-r. Sadeghi, and T. Holz, "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)*, 2015.

[27] 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.

[28] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level," in *Proceedings of IEEE Symposium on Security and Privacy (Oakland'16)*, 2016.

[29] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, "VTrust: Regaining Trust on Virtual Calls," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[30] 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.