Devil is Virtual: Reversing Virtual Inheritance in C++ Binaries

Rukayat Ayomide Erinfolami
Binghamton University
rerinfo1@binghamton.edu

Aravind Prakash
Binghamton University
aprakash@binghamton.edu

ABSTRACT

The complexities that arise from the implementation of object-oriented concepts in C++ such as virtual dispatch and dynamic type casting have attracted the attention of attackers and defenders alike. Binary-level defenses are dependent on full and precise recovery of class inheritance tree of a given program. While current solutions focus on recovering single and multiple inheritances from the binary, they are oblivious of virtual inheritance. The conventional wisdom among binary-level defenses is that virtual inheritance is uncommon and/or support for single and multiple inheritances provides implicit support for virtual inheritance. In this paper, we show neither to be true.

Specifically, (1) we present an efficient technique to detect virtual inheritance in C++ binaries and show through a study that virtual inheritance can be found in non-negligible number (more than 10% on Linux and 12.5% on Windows) of real-world C++ programs including Mysql and Libstdc++. (2) we show that failure to handle virtual inheritance introduces both false positives and false negatives in the hierarchy tree. These false positives and negatives either introduce attack surface when the hierarchy recovered is used to enforce CFI policies, or make the hierarchy difficult to understand when it is needed for program understanding (e.g., during decompilation). (3) We present a solution to recover virtual inheritance from COTS binaries. We recover a maximum of 95% and 95.5% (GCC -O0) and a minimum of 77.5% and 73.8% (Clang -O2) of virtual and intermediate bases respectively in the virtual inheritance tree.

CCS CONCEPTS

• **Binary Analysis → Virtual inheritance; Class inheritance recovery**; • **Software Security → CFI**.

KEYWORDS

Virtual inheritance recovery, Class inheritance recovery, software reverse engineering

ACM Reference Format:
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 (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3372297.3417251

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417251

1 INTRODUCTION

Recovering high-level semantic information from binaries has strong security relevance in areas such as vulnerability detection, control-flow integrity (CFI) [2, 5, 6, 8, 17, 34, 38, 42, 44], decompilation [3, 7, 10, 12, 21, 40, 41] and memory forensics [15]. In particular, recovery of object-oriented semantics (e.g., class hierarchy) is key to C++ binary-level defenses (e.g., [11, 14, 17, 31, 33]).

Traditional C++ binary analysis solutions have focused on constructor analysis [8, 9], destructor analysis [16], overwrite analysis [31], and VTable analysis [17, 33, 43] in order to recover at least a partial class hierarchy tree (CHT). While prior solutions have focused on recovering single and multiple inheritances in the binary, virtual inheritance—an important feature of C++ language—has been ignored. From a security standpoint, some key questions arise: 1) How common is virtual inheritance? 2) Is virtual inheritance relevant for security? 3) Does support for single and multiple inheritance implicitly cover virtual inheritance?

Virtual inheritance is not uncommon: Virtual inheritance in C++ facilitates the implementation of key design ideas, and has been used in prominent and widely-used programs (e.g., Libstdc++, Mysql). Our first study comprising of 1129 Linux C++ binaries found 11% of the libraries to contain virtual inheritance while our second study of 648 Windows binaries found 12.5% with virtual inheritance. Widely used libraries such as Libstdc++ utilize virtual inheritance to prevent duplication of stream objects in the IO-related classes. Because Libstdc++ is linked to all C++ programs, virtual inheritance can be commonly found in most C++ programs' memory.

Security relevance of virtual inheritance: Failure to handle virtual inheritance results in severe security flaws. Current binary-level CFI defenses against C++ virtual dispatch attacks extract the VTables in a binary and given a callsite, they construct a policy that allows the callsite to target a strict subset of polymorphic virtual functions derived from the class inheritance tree. Without specific mechanisms to handle virtual inheritance, current solutions either suffer from false negatives or false positives in the inheritance tree. In the case of Marx [31], compiler-generated "construction VTables" (transient VTables used in construction of objects with virtual bases) are incorrectly included in the inheritance tree as regular VTables (i.e., VTables that represent a class) thereby resulting in false positives in the inheritance tree. Whereas, in the case of VCI [9] legitimate inheritance relationships arising due to virtual inheritance are completely missed due to the lack of support for virtual inheritance. VCI is testament to the fact that *support for single and multiple inheritance does not implicitly cover virtual inheritance.* Both false positives and negatives result in inaccuracies in resulting CFI policies.

Unlike single and multiple inheritances, the recovery of virtual inheritance poses significant technical challenges. First, thanks to the *is-a* property, reference to a derived object is also a legitimate reference to its virtual base object. However, by definition, a single copy of the virtual base is retained in the entire inheritance tree. As such, offset of the virtual base sub-object from a derived class object and an intermediate class sub-object (i.e., object of a class between the derived class and the virtual base class in the inheritance tree) could be different. Any binary-level static object-layout analysis that intends to capture virtual bases must take into account various offsets from different derived objects in an inheritance tree. Second, the ABI [1] necessitates additional structures and fields, e.g., virtual base offset (vbase-offset), virtual call offset (vcall-offset), construction VTable, VB-Table, etc. in order to implement virtual inheritance. These fields and structures introduce complexities in implementation that require special handling. Finally, virtual bases are allocated at the end of all the non-virtual bases in an inheritance (sub) tree. It is therefore important for a virtual inheritance recovery solution to delineate between non-virtual and (one or more) virtual bases in an object's memory.

In this paper, we first show that virtual inheritance is not uncommon. To this end, we perform a comprehensive study of C++ binaries in the default installation of Ubuntu Linux 18.04 distribution and report that 11% of C++ libraries contain virtual inheritance. We also performed a study of Windows C++ DLLs and report that 12.5% of them contain virtual inheritance. Further, we design a robust virtual inheritance recovery engine that pivots on the ABI definitions (both Itanium for gcc and clang, and MSVC for Microsoft Visual Studio). Our solution is tolerant to compiler variations including optimizations. Our class inheritance engine codenamed `VirtAnalyzer` employs object-offset analysis that can identify virtual bases in a derived object with a high level of precision. `VirtAnalyzer` is able to successfully recover a maximum of 95% and 95.5% (GCC -O0) and a minimum of 77.5% and 73.8% (Clang -O2) of virtual and intermediate bases respectively in the virtual inheritance tree.

Our contributions can be summarized as follows:

(1) We present simple and efficient algorithms to detect the presence of and recover virtual inheritance in a given C++ binary that adheres to either Itanium or MSVC ABI. Our techniques are ABI-based, and so are largely unaffected by the specific compiler and/or optimizations (except in cases where entire classes are removed by the compiler).
(2) We show that virtual inheritance is not uncommon in C++ binaries with significant security concerns. It cannot be ignored.
(3) We present a sample attack that depicts how false positives in the CHT due to virtual inheritance can be exploited despite state-of-the-art defenses. We further demonstrate that an exponential (O(n^2)) attack surface manifests where n is the depth of the inheritance subtree with a virtual base.
(4) `VirtAnalyzer` is available at https://github.com/bingseclab/VirtAnalyzer

2 TECHNICAL BACKGROUND

In this section, we provide the technical details needed to understand the remainder of the paper. Because the Itanium ABI is widely used (adhered to by gcc and clang) and the specification is openly available [1], we use it as a focal point of our work. However, our work also supports MSVC ABI (see Section 6.3).

Listing 1: Running example

```
class A{
public:
int a;
virtual void af(){...}
};
class B: public virtual A
    {
public:
int b;
virtual void bf(){...}
};
```

```
class C: public virtual A
    {
public:
int c;
virtual void cf(){...}
};
class D: public B, public
    C{
public:
int d;
virtual void df(){...}
};
```

Listing 2: Disassembly of the constructor of D in the running example.

```
...
mov [rbp+var_8], rdi
mov rax, [rbp+var_8]
add rax, 20h
mov rdi, rax; this, at offset 20h
call _ZN1AC2Ev; A::A(void)
...
mov rax, [rbp+var_8]
lea rdx, off_201BB8; subVTT address of B-in-D
mov rsi, rdx
mov rdi, rax; this, at offset 0
call _ZN1BC2Ev; B::B(void) primary base class
...
mov rax, [rbp+var_8]
add rax, 10h
lea rdx, off_201BC8; subVTT address of C-in-D
mov rsi, rdx
mov rdi, rax; this, at offset 10h
call _ZN1CC2Ev; C::C(void)
...
```

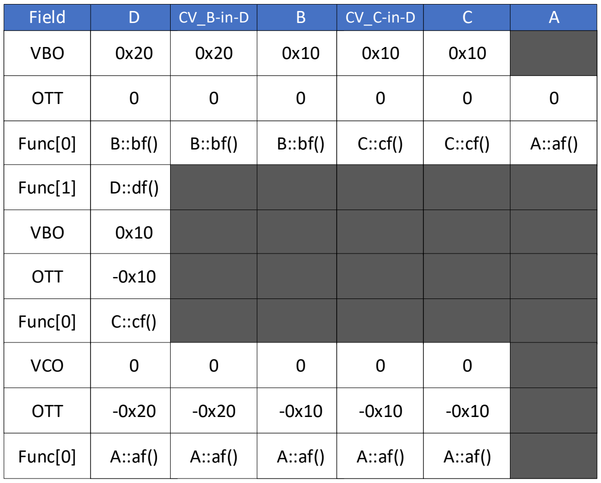
2.1 Running Example

We will use the running example in Listing 1 throughout the paper. Class A is inherited virtually by classes B and C. Class D inherits from classes B and C to form what is popularly known as the "diamond" structure. Because class A is inherited virtually, only one copy of the sub-object of A is retained in the object of class D. Listing 2 shows the disassembly of D's constructor. All code examples in this paper were compiled using GCC 7.3 with optimization flag O0, except otherwise stated

Definitions of key terms used in this paper are reproduced in Appendix A for readers' convenience.

2.2 Virtual Inheritance

Virtual inheritance is the solution to the "diamond" structure problem, wherein multiple inheritance results in multiple copies of a base class' member variable(s) in the object of a derived class. In Listing 1, since B and C virtually inherit from A, the compiler is instructed to keep a single copy of A in D. The object of D is such that there is exactly one copy of A's sub-object, which is placed at the



Field	D	CV_B-in-D	B	CV_C-in-D	C	A
VBO	0x20	0x20	0x10	0x10	0x10	■
OTT	0	0	0	0	0	0
Func[0]	B::bf()	B::bf()	B::bf()	C::cf()	C::cf()	A::af()
Func[1]	D::df()	■	■	■	■	■
VBO	0x10	■	■	■	■	■
OTT	-0x10	■	■	■	■	■
Func[0]	C::cf()	■	■	■	■	■
VCO	0	0	0	0	0	■
OTT	-0x20	-0x20	-0x10	-0x10	-0x10	■
Func[0]	A::af()	A::af()	A::af()	A::af()	A::af()	■

Figure 1: VTable fields of classes in the running example. “CV": construction VTable, VBO: vbase-offset, OTT: offset-to-top

end of D’s object. Virtual inheritance is achieved by prefixing the base class name in the class signature with the keyword “virtual".

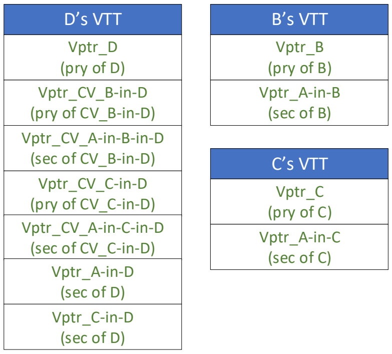
Virtual Base Offset Every class which inherits from a virtual base, either directly or indirectly, has a “vbase-offset". It is the offset of a virtual base sub-object from a derived sub-object. This value is used when a member variable in the virtual base sub-object needs to be accessed from a pointer to a derived object. It is also used during the initialization of the secondary VTable corresponding to a virtual base. The VTable of a class has a vbase-offset field for each of its virtual bases. For instance, as shown in Figure 1, D has two vbase-offsets of values 0x20 (offset from D’s sub-object to A-in-D’s sub-object) and 0x10 (offset from C-in-D’s sub-object to A-in-D’s sub-object).

Construction VTable Construction VTables are used during construction and destruction of intermediate bases in a virtual inheritance tree. They are needed to access the correct vbase-offset and virtual functions associated to a given base. Consider the running example and Figure 1, B needs to be constructed in D. If B’s VTable is used for the construction of B in D, the vbase-offset that will be retrieved is 0x10 (offset to A’s sub-object in B). However, the vbase-offset of D (shared with B) is 0x20. Retrieving a vbase-offset of 0x10 instead of 0x20 will result in accessing the wrong location in D’s object as A’s sub-object. Therefore, there is the need for another VTable corresponding to B-in-D that has the correct vbase-offset(0x20) and the virtual functions associated with B (this is because while constructing B’s sub-object in D, B’s virtual functions should be accessible not those of D), as well as special constructors and destructors which access the construction VTables.

Figure 1 shows the fields of the two construction VTables in the running example (CV_B-in-D, CV_C-in-D). Every intermediate base class has an associated construction VTable. As shown in Table 1, the construction VTable of an IntermediateBase-in-Derived has the vbase-offset, vcall-offset and offset-to-top of virtual bases associated with the derived, while the virtual functions, type info and offset-to-top of non-virtual bases associated with the intermediate base.

Table 1: Differences and similarities among the fields of a VTables and a Construction VTable using running example

Fields	Construction VTable of B-in-D	VTable of B
vbase-offset	higher e.g 0x20 in B-inD	lower e.g 0x10 in B
offset-to-top (non-virtual subVTable)	same	same
offset-to-top (for virtual subVTable)	lower e.g -0x20 for A’s VTable in B-in-D	higher e.g -0x10 for A’s VTables in B
vcall-offset	lower e.g -0x20 for af() in B-in-D	higher e.g -0x10
type-info	Same	Same
Function pointers	Same	Same



D’s VTT
Vptr_D (pry of D)
Vptr_CV_B-in-D (pry of CV_B-in-D)
Vptr_CV_A-in-B-in-D (sec of CV_B-in-D)
Vptr_CV_C-in-D (pry of CV_C-in-D)
Vptr_CV_A-in-C-in-D (sec of CV_C-in-D)
Vptr_A-in-D (sec of D)
Vptr_C-in-D (sec of D)

B’s VTT
Vptr_B (pry of B)
Vptr_A-in-B (sec of B)

C’s VTT
Vptr_C (pry of C)
Vptr_A-in-C (sec of C)

Figure 2: The VTT layout of classes in running example.“CV" means construction VTable, “pry" means primary VTable, “sec" means secondary VTable

Virtual Table Table The virtual table table (VTT)[1] is an array of VTable and construction VTable pointers (if any exist) of a class (Figure 2). Every derived class with at least one direct or indirect virtual base class(es) has an associated VTT. The VTT is made up of pointers to the primary and seconday VTables of the derived class, and the primary and secondary construction VTables of its intermediate base classes. We refer to each complete object VTable pointer (i.e. group of primary and secondary VTable pointers of a class) within a VTT as a SubVTT. Basically, a VTT is made up of multiple subVTTs each pointing to a complete object VTable or a construction VTable. Only one of those subVTTs point to the VTable of the derived class which owns the VTT, the others point to construction VTables.

As mentioned in Section 2.2, there is the need for special constructors (and destructors) to construct intermediate base sub-objects. The constructor of the derived class passes a pointer to the subVTT corresponding to the construction VTable of the IntermediateBase-in-Derived to the special constructor as a second hidden argument.

[1] VCI [9] uses the acronym VTT to refer to VTable group, which is different from the Virtual Table Tables defined in the Itanium ABI [1]. In this paper, we stick to the terminology used in the ABI.

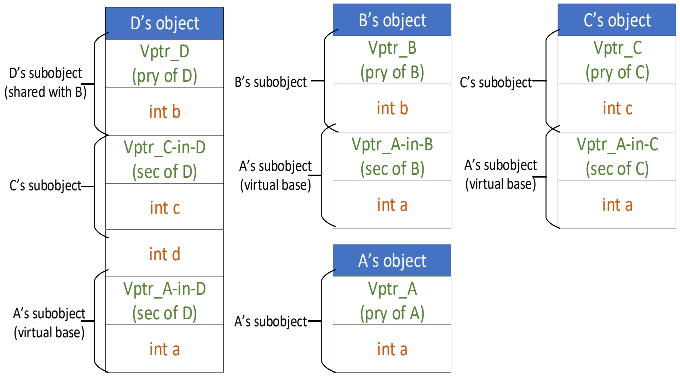


Figure 3: Object layout of classes in the running example.

It then initializes the intermediate base sub-object by accessing the subVTT (Listing 2, lines 7 and 13) as opposed to initializing with immediate values in the case of single and multiple inheritance.

Order of Object Construction We will explain this using the running example and Listing 2. First, the constructor of D constructs all its virtual bases, A in this case. Next, it constructs B by calling the special constructor of B. The same is done for C. Finally, the vptrs of D are written into appropriate locations in the object starting with the primary vptr.

Object and VTable Layout with Virtual Inheritance Figure 3 shows objects of types A, B, C and D. An object of D also contains a sub-object of A. Note that none of the objects share sub-object with A, since the sub-object of A could be shared among multiple bases. This is also the same for VTables Figure 1, a derived class does not share VTable with its virtual base (except the virtual base is either empty or nearly empty). Note that only the VTable corresponding to the virtual base has vcall-offset fields.

3 SECURITY IMPACT OF VIRTUAL INHERITANCE

3.1 Study: Virtual Inheritance in Real-World Programs

Virtual inheritance in C++ has received very little attention. Prior efforts have focused on single and multiple inheritances, therefore, support for handling virtual inheritance is missing in both source-code-level solutions[4, 14, 19, 20, 23, 24, 28, 29, 37, 42] and binary-level solutions[9, 16, 26, 31, 39]. While it is true that virtual inheritance in C++ is not as common as single or multiple inheritance, we conducted a study in order to understand how (un)common virtual inheritance is in realworld programs. We evaluated 1129 C++ binaries on Ubuntu 18.04 Linux distribution and found 11% of the libraries with instances of virtual inheritance ranging from 1 to 27. For Windows distribution, we evaluated 648 DLLs and found 12.5% with instances of virtual inheritance ranging from 1 to 382. Our approach for detecting virtual inheritance in a binary is described in detail in Sections 6.1 and 6.2. Findings of our study are tabulated in Table 2. Notably, we found that virtual inheritance is prevalent in both libraries and executables including the Libstdc++ library and Mysql database engine.

State-of-the-art binary analysis tools that rely on inference of class hierarchy like Marx[31] and VCI[9] do not recover virtual inheritance, which is necessary for enforcing precise CFI policies.

Table 2: Prevalence of virtual inheritance in C++ programs

ABI		# C++	# with Virtual Inheritance
Itanium	Libraries	219	26
	Executables	910	19
MSVC	DLLs	648	81

3.2 False Positives and False Negatives in State-of-the-art Binary Level Solutions

We consider state-of-the-art binary analysis tools which reconstruct high level semantics from the binary. Table 3 shows the weaknesses (false positives or false negatives) of the solutions in handling virtual inheritance. As representative solutions, we also provide a detailed description of how VCI [9], SmartDec [16] and MARX [31] behave when virtual inheritance is present in a binary.

Table 3: Binary level solutions which recover high level semantic information from the binary

Solution	Introduces False +ve	Introduces False -ve	Distinguishes Construction VTables from Regular VTables
VCI[9]	✗	✓	✗
Marx[31]	✓	✗	✗
SmartDec[16]	✗	✓	✗
OOAnalyzer[35]	✗	✗	✗
vfGuard[33]	✓	✗	✗
Katz et al.[26]	✓	✗	✗
ROCK[27]	✓	✗	✗
ObjDigger[25]	✓	✗	✗
Lego[39]	✓	✗	✗
Hex Rays[36]	✗	✓	✗
BinCFI[45]	✓	✗	✗
VTint[43]	✓	✗	✗
Our Analysis	✗	✗	✓

False Positives in Marx.

Marx Overview: Marx is a binary-level solution that defends against abuse of virtual dispatch mechanism in C++. In a nutshell, for a given C++ virtual callsite, Marx identifies all the polymorphic functions that can be invoked at that callsite, and instruments the binary to allow only those functions. Allowable polymorphic functions are recovered by performing overwrite analysis which identifies sets of vptrs that get overwritten in an object during construction or destruction. Marx groups classes into sets wherein each set represents a class inheritance sub-tree with no particular inheritance order. A full description of Marx can be found at [31].

For the running example, Marx recovers six complete-object VTables (including construction VTables). One VTable each for A, B, C and D and one construction VTable each for B-in-D and C-in-D. Marx does not make any distinction between VTables and construction VTables. In other words, Marx will interpret construction VTables to be representations of legitimate classes in the binary. For the running example, 12 VTables (breaking them into primaries and secondaries) are recovered. Under O0 optimization, Marx groups

Listing 3: Source code for main function

```
int main(){
                B *b = new B();
                b->bf();
                return 0;
}
```

Listing 4: Disassembly for function B::bf()

```
...
1. mov rax, [rbp+var_8] ; get object address
2. mov ecx, [rax+8]; get member at offset 0x8
3. mov rax, [rax] ; get object's VTable
4. sub rax, 18h ; locate vbase-offset field
5. mov rdx, [rax] ; get vbase-offset
...
6. mov rax, [rbp+var_8] ; get object addres
7. add rax, rdx ;locate virtual base (VB) sub-object
8. mov edx, [rax+8]; get virtual base member
...
9. add edx, ecx
10. mov [rax+8], edx ; write result to VB sub-object
```

all 12 into a single set, while for O2, there are 3 sets. The sets for O2 are:

- set1 = $\{CV\_B - in - D_{pry}, B_{pry}, D_{pry}\}$
- set2 = $\{CV\_C - in - D_{pry}, C_{pry}, D_{sec}\}$
- set3 = $\{CV\_B - in - D_{sec}, B_{sec},$ $CV\_C - in - D_{sec}, C_{sec}, D_{sec}, A_{pry}\}$

Set1 shows that Marx will incorrectly allow the vptr of Construction VTable B-in-D at a callsite that expects an object of type B. Consider listing 3 and listing 4 which show a call to function B::bf() and the disassembly of function B::bf() respectively. B::bf() simply adds up member variable b of class B with the member variable a of class A and writes the result back to a. If an attacker successfully overwrites the vptr of the object of B with that of the construction VTable B-in-D before the callsite, Marx would not raise any alarm.

We see from Table 1 that the vbase-offset in the construction VTable B-in-D is greater than that in the VTable of B. If the construction VTable of B-in-D is used, line 4 of listing 4 retrieves a vbase-offset equal to 0x20 which is used to access the virtual base sub-object at line 7. This is outside the bounds of B's object since the total size of B is 0x20. Lastly, an offset of 0x28 (0x20 + 0x8) from B's object is accessed at line 8 to get a member of the virtual base, and written in line 10. These read and write operations occur outside B's object bounds. The failure of Marx to handle virtual inheritance (meaning that construction VTables are neither identified nor filtered out from the inheritance sets) introduces an attack surface for data corruption and arbitrary code execution under favorable circumstances. A detailed PoC attack is provided in Section 4.1.

False Negatives in VCI and SmartDec.

VCI and SmartDec Overview: VCI and SmartDec are binary level class hierarchy recovery tools which attempt to reason about the direction of inheritance. VCI achieves this by performing constructor only analysis, while SmartDec performs constructor and destructor analysis. Both solutions analyze the order in which base class constructors and/or destructors are called from the derived class constructor/destructor. The analysis is done by simply scanning constructors, for instance, for assembly callsites that invoke other constructors. While VCI puts measures in place to filter out composed classes from the class hierarchy tree, SmartDec includes both composed and inherited classes in the class hierarchy tree.

Listing 5: Disassembly for main function

```
1.  call _Znwm
2.  mov rbx, rax
...
3.  call _ZN1DC1Ev; D::D(void)
4.  mov [rbp + var_18], rbx
5.  mov rax, [rbp+ var_18]
6.  mov rax, [rax]; deref this ptr to get vptr
7.  mov rax, [rax]; deref vptr to get addr of bf()
8.  mov rbx, [rbp+ var_18]
9.  mov rdi, rax
10. call rax
```

Regular constructors are known to be functions which initialize vptrs as immediate values. As mentioned in Section 2.2, special constructors used to construct intermediate bases in a virtual inheritance tree do not initialize any vptr using immediate values. Considering the disassembly in Listing 2, VCI and SmartDec will recover A as the direct base of D and ignore B and C since they are unable to identify any calls to the constructors of B and C from the constructor of D. VCI keeps a metadata in the binary which maps every function to the set of class types it can be invoked on, therefore for each function in the example, a map that looks like this is generated: af() = {A,B,C,D}, bf() = {B}, cf() = {C}, df() = {D}. Say function bf() is to be invoked on an object of type D as shown in Listing 5, before the callsite at line 10, VCI checks if there is a class in the set for bf() which has a vptr equal to the vptr obtained at line 6. Since the vptr belong to D and D is not in the set, VCI raises a false violation alarm. SmartDec will behave similarly.

4 EXPLOITING VIRTUAL INHERITANCE

4.1 Defeating Marx

In this section we present a proof-of-concept attack launched against a synthetic vulnerable program, Listing 6. The victim program is hardened with an implementation of the Marx VTable protection policy (only the Marx hierarchy recovery tool is open source, not the VTable protection tool). This policy ensures that only virtual functions from the set of classes related to the callsite type are allowed at runtime. The attack is successful because Marx does not differentiate between regular and construction VTables.

Attack Model The objective of this attack is to execute arbitrary code. We assume that the attacker can bypass Address Space Layout Randomization (ASLR) and Stack Protector.

The first assumption allows the attacker to identify the absolute addresses of suitable construction VTables to use in the attack. Bypassing ASLR to reveal such information is possible as shown in the literature [13, 22]. Once the address of a suitable construction VTable has been found, there is a need to write it into appropriate location in the object. Our PoC exploits buffer overflow vulnerability for this which is possible by bypassing stack protector. There are works that show that bypassing stack protector is possible , for this reason we simply disable stack protector for this PoC.

PoC Attack: Arbitrary Code Execution The victim program has a class hierarchy similar to the running example, with additional classes A1 and A2 which are proper base classes of A. In the main

Listing 6: Vulnerable program

```
1.  class A1{
     virtual int geta1(){...}
2.  };
3.  class A2{ ...
4.   virtual int geta2(){...}
5.   virtual void execShell(){system("/bin/sh");}
6.  };
7.  class A: public A1, public A2{ ...
8.   virtual int geta(){...}
9.  };
10. class B: public virtual A{ ...
11.  virtual int callBaseFunc(){return geta();}
12. };
13. class C: public virtual A{ ...
14. };
15. class D: public B, public C{ ...
16. };
17. int main(){ ...
18.  B b;
19.  char buf2[10];
20.  char buf1[20];
21.  scanf("%20s", buf1);
22.  strcpy(buf2, buf1); //overflow buf2
23.  b.callBaseFunc();
24.  return 0;
25. }
```

Listing 7: Disassembly of B::callBaseFunc()

```
...
1. mov eax [b_obj_addr] ;get vptr of B
;Marx-like check: is_vptr_valid(eax) -> True
2. sub eax, 0xC ;locate vbase-offset field in VTable of B
;get vbaseoffset, 0x20 instead of 0x10 after overflow
3. mov eax, [eax]
4. ...
5. add eax, b_obj_addr ;reach sub-object A2 instead of A
6. mov eax, [eax] ;get vptr of A2 instead of vptr of A
;get second func in VTable, execShell() instead of geta()
7. add eax, 4
8. ...
9. call eax
```

function, an object of class B is created. The main function has a buffer overflow vulnerability on line 22. On line 23, callBaseFunc() is called on B's object which in turn calls function geta() defined in A (B's primary base). Instead of this intended call, the PoC attack diverts control to function execShell() defined in A2 (B's secondary base). Note that A's sub-object in B is located at offset 0x10 from the top of B's object, while A2's sub-object is located at offset 0x20.

First, we identified the address of the construction VTable of B-in-D which has a vbaseoffset of 0x20. Next, we exploited the buffer overflow vulnerability to corrupt the address point of object b. We overflow buf2 (line 22) into b such that the vptr of B (which has a vbaseoffset of 0x10) is overwritten with that of the construction VTable of B-in-D. Recall that Marx will allow this since it does not differentiate between a regular and a construction VTable. We show the disassembly of function B::callBaseFunc() in Listing 7. On line 3 of Listing 7, the vbaseoffset is retrieved to locate A's subject in B. Because of the buffer overflow, a vbaseoffset of 0x20 is retrieved thereby locating the sub-object of A2 instead. As a result, line 7 locates the second virtual function in the VTable of A2 (execShell) and executes it.

4.2 Attack Surface Analysis

The attack surface that manifests due to the presence of virtual inheritance directly relates to the number of construction VTables. That is, the number of offsets that can be exploited increases with the number of construction VTables present in the binary, especially if they contain sufficient unique offsets. Unique vbase-offset and offset-to-top values for representative realworld programs in our study is presented in Figure 4. Results indicate that it is not uncommon for offset values to be in multiple hundreds, which in turn indicates potential for an attacker to perform memory corruption attacks multiple hundred bytes from an intended access.

In general, attack surface increases with the number of construction VTables, which in turn increases *exponentially* with depth of inheritance. Table 4 presents the number of construction VTables for the running example (row 1) and increasing depths (up to n). Total number of construction VTables at depth n is:

$$\sum(n+1) - 1 = \frac{(n+1)(n+2)}{2} - 1 \implies O(n^2) \qquad (1)$$

Since each derived class in the virtual inheritance tree may have several varying object layouts, there is a high probability that an attacker will find sufficient unique offsets needed to carry out an attack. For instance, in Mysqld, there are 6 unique virtual bases in the entire inheritance tree, however, we found 24 unique vbase-offsets and 24 unique offset-to-top values. Furthermore, in program's that use libraries such as Libstdc++, inheriting from virtual bases (e.g., Stream class) will introduce additional construction VTables in the program's memory.

Table 4: Table showing how the number of construction VTables increase with inheritance depth

Inheritance tree	# of construction VTables
As in Listing 1 (depth 1)	2
E inherits from D (depth 2)	2+3 = 5
F inherits from E (depth 3)	2+3+4 = 9
X inherits from Y (depth n)	2+3+4+...+n+1 = $\sum(n+1) - 1$

5 SOLUTION OVERVIEW

An overview of our solution is presented in Figure 5. `VirtAnalyzer` tackles multiple challenges posed by virtual inheritance.

5.1 Challenges in Recovering Virtual Inheritance

Presence of Optional Fields. Unlike single and multiple inheritance where mandatory fields such as offset-to-top and type-info (i.e., RTTI) provide a reliable signature, virtual inheritance introduces *one or more* optional fields, vcall-offsets and vbase-offsets, which makes analysis difficult. These fields pose 2 main challenges. First, without knowing how many optional entries are present, identifying the boundaries of VTables is hard. Second, because one or more entries of vcall-offsets could be laid immediately preceding one or more entries of vbase-offsets, it is hard to demarcate the end of vcall-offsets and the beginning of vbase-offsets.

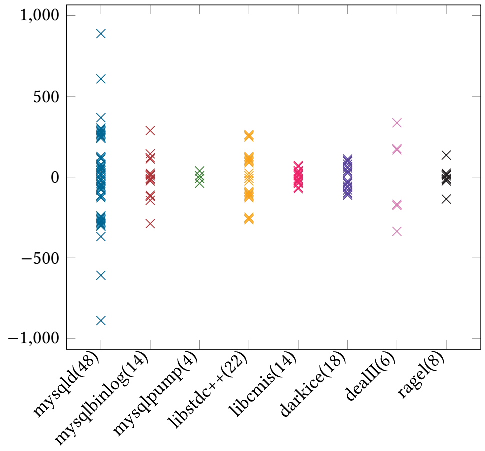


Figure 4: Distribution of vbase-offset and offset-to-top from construction VTables. The number in parentheses next to the binary name is the exact number of unique vbase-offset and offset-to-top values for that binary.

Construction VTables vs Regular VTables. The layouts of a construction VTable and a regular VTable are exactly the same. However, they have different purposes. Therefore a trivial signature-based approach is insufficient to distinguish between the two. Moreover, a class may contain multiple construction VTables depending on the depth of inheritance between the virtual base and the derived class. In order to build a clear and accurate class hierarchy tree, we must be able to group all the construction VTables of a class and associate them with the complete object VTable of that class.

Differentiating Virtual and Non-virtual Bases. When a class derives from both virtual and non-virtual bases, its object and its VTable contain sub-objects and subVTables that correspond to both bases. In order to correctly reconstruct the virtual inheritance tree, there is the need to identify non-virtual bases and filter them out.

5.2 High-Level Approach

The Figure 5 shows the overview of `VirtAnalyzer`. It incorporates analysis passes that tackle the challenges described above.

Discerning Relationship Between Mandatory and Optional Fields. There is simply not enough information in one VTable alone to demarcate different optional and mandatory fields. Therefore, our analysis combines information in multiple VTables. For instance, the offset-to-top value from a secondary VTable corresponding to a virtual base is equal in magnitude to the vbase-offset in the derived object's primary VTable. Such a correlation provides a strong confirmatory test to filter vbase offsets. Additionally, when the types are statically known during compilation, a vbase-offset is applied by the compiler during computation of vbase object's address from a derived object's address. By cross referencing potential offsets in the VTable with offseting code emitted by the compiler, it is possible to identify optional offsets with a high level of confidence.

Identifying Construction VTables. `VirtAnalyzer` incorporates VTT analysis in order to identify construction VTables. The VTT is a key signifier of virtual inheritance in a binary, however, it is also crucial for differentiating construction VTables from regular VTables. Per ABI mandate, the first entry in a VTT always points to a regular VTable, and every other entry points to a construction VTable. `VirtAnalyzer` first identifies VTTs in the binary and then isolates the regular and construction VTables from the first and remaining entries in a VTT.

Grouping Construction VTables of a Class. The virtual function fields in all construction VTables of a class and its regular VTable are exactly the same. `VirtAnalyzer` takes advantage of this similarity to identify and group all the construction VTables belonging to a class.

Identifying Virtual and Non-virtual bases. Only virtual bases have associated optional fields such as vbase-offset and vcall-offset. Since `VirtAnalyzer` has already recovered the optional fields in an earlier step, it can filter out non-virtual bases.

6 VIRTANALYZER

`VirtAnalyzer` consists of two phases, each of which consists of multiple sub-phases. We show the data used in each sub-phase in Table 10. We explain the phases and their sub-phases below.

6.1 Phase 1: Extracting Metadata from the Binary

In this phase, we recover information in the binary which indicate virtual inheritance. The sub-phases here include identifying certain structures, values and functions, such as VTables, VTTs, subVTTs, vbaseoffsets, constructors and destructors.

Identifying VTables We identify VTables by implementing algorithms presented in vfGuard [33] and DeClassifier [11]. Usually, VTables are referenced from the text section using immediate values that point to the read-only section. However, we found some libraries with immediate values pointing to the got section, which then point to the VTables. Such cases too were handled in our analysis. We recover immediate values and examine them to see if they are vptrs. An immediate value is a vptr if all 3 of the following conditions hold. That is, (a) the vptr points to a function start address or the pure virtual function, (b) (vptr - DWORD_SIZE) contains either zero, or points to the data section (the typeinfo), and (c) (vptr - DWORD_SIZE*2) contains zero (for a primary VTable) or a negative value (for a secondary VTable).

We identify valid vptrs and then group primary vptrs with their corresponding secondary vptrs to obtain the complete object VTable of each class. This was done by implementing the VTable grouping algorithm introduced in DeClassifier [11]. This set of VTables also includes construction VTables.

Identifying VTTs VTTs, like VTables also reside in the read-only section of the binary. They are the only structures whose entries are pointers to VTables. We identify VTTs by first identifying structures that contain at least two entries of pointers to known VTables.

Unlike VTables whose offset-to-top field or typeinfo field can be used to separate two VTables which are laid out contiguously, VTTs only contain pointers, as a result, it is tricky to identify the boundaries between VTTs if they are laid out contiguously. In most cases, the VTables pointed to by a VTT are laid out immediately after the VTT which makes it easy to know where a VTT ends.

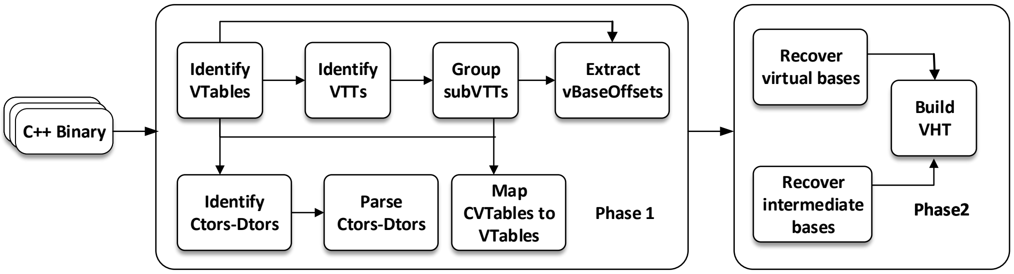


Figure 5: Overview of `VirtAnalyzer`

However, we found a few cases where VTTs of different classes are laid out contiguously. In these cases there is the possibility of wrongly grouping those multiple VTTs as one VTT. To address this, we take advantage of how VTT entries are ordered.

Notice from Figure 2 that the VTT of D starts with the primary vptr of D, followed by all the construction VTables (i.e. the second entry points to a construction VTable), and ends with D's secondary VTables. We also noticed that construction VTables are laid out after their Derived VTables. We first store the values of the first and second entries of a VTT and continue down the VTT. We say that the previous entry found is the last entry in a VTT if the value of the current entry is either less than the first entry (the last entry should be greater than the first since it belongs to a secondary VTable) or greater than the second entry (the last entry should be less then the second since construction VTables are laid out after the derived VTable). Algorithm 1 shows how VTTs are identified.

Grouping subVTTs A VTT has pointers to a VTable and one or more construction VTables. We refer to a group of pointers to each of these complete object VTables as subVTT. The first subVTT in a VTT points to the regular VTable of the derived class, while the other subVTTs point to its construction VTables. We differentiate regular VTables from construction VTables by looking at their subVTT positions in the VTT. To identify subVTTs, we start scanning a VTT from the beginning. Anytime we find a pointer to a primary VTable, we create a new subVTT, and all secondary VTable found (before the next primary VTable) are grouped together. We are able to prevent grouping the secondary VTables of the derived class with the last construction VTable because the secondary VTables' addresses are less than those of the construction VTables. Algorithm 2 presents the specific steps in grouping subVTTs.

Extracting Virtual Base Offsets There is the possibility of recovering false VTables and this will result in recovering false VTTs. A group of pointers may point to recovered false VTables, we will wrongly identify this as a VTT. However, by recovering and verifying the vbase-offset they contain, we will realize that they are invalid. The vbase-offset of a class is also used in recovering its virtual bases. Algorithm 3 shows the steps used in this sub-phase.

The vbase-offset is one of the optional fields a VTable may contain depending on whether it is in a virtual inheritance tree or not. The number of vbase-offsets a derived class has is equal to the number of its virtual bases. Since a VTable can contain multiple vbase-offsets, we recover the first vbase-offset by deferencing the third Dword from the vptr (upward direction) and then we go up the VTable one Dword at a time to recover the rest. As we recover each vbase-offset, we verify it. This is done by comparing it with the offset-to-top values in the secondary VTables from the first to the last. If the current vbase-offset is not equal to the negative value of the offset-to-top of current secondary VTable, we assume the base corresponding to that VTable is not a virtual base, then we move to the next secondary VTable. We stop scanning for vbase-offset when the number of Dwords above the vptr checked is equal to the number of secondary VTables. We must find at least one matching vbase-offset and offset-to-top to conclude that the subVTT is valid. If we do not find any valid subVTT in a VTT, we discard the VTT.

Mapping Construction VTables to Regular VTables As stated in Section 2.2, only the first subVTT in a VTT belong to the VTable of the derived class, the other subVTTs belong to construction VTables of IntermediateBase-in-Derived. We use this information to differentiate a construction VTable from a regular VTable. In this sub-phase, we identify the corresponding complete object VTable of every construction VTable. One or more construction VTables can map to one complete object VTable. For instance, in the running example, we map the construction VTable of B-in-D to the VTable of B. The mapping is constructed based on the observation that the function pointers in a complete object VTable and its corresponding construction VTables are exactly the same. To construct the mapping, we sum up the function pointers in each VTable and use that as the key in a dictionary. A regular VTable and its corresponding construction VTables will have the same key. This mapping is needed while building the virtual inheritance tree.

Identifying Constructors and Destructors Constructors and destructors are where vptr initializations take place, for both object construction and destruction. We identify constructors and destructors, using the same approach employed by existing solutions. A function is said to be a constructor or destructor if it initializes an immediate value which points to a known VTable.

Parsing Constructors and Destructors We analyze each instruction within constructors and destructors to keep track of how offset in an object and VTT addresses are propagated from one register or memory location to another. We use this information to identify and retrieve the hidden arguments (`this` pointer and/or subVTT address) passed to regular constructors and destructors as well as the special constructors and destructors. We also keep track of call instructions which will be used in phase 2 of the analysis to identify virtual and intermediate bases.

6.2 Phase 2: Recovering Bases

Virtual inheritance can either be direct (i.e., a base of a derived class is virtually inherited) or indirect (i.e., an intermediate base class of a derived class virtually inherits from its base). Specific details can be found in section 2.5.3 of the ABI [1]. Further, it is possible that a derived class has a mix of virtual and non-virtual bases.

Recovering Virtual Bases The constructor of a derived class directly calls the constructors of its virtual bases, be it direct or indirect, along with the constructors of its direct non-virtual bases. Before calling the constructor of a virtual base, an offset equal to the vbase-offset corresponding to that base is added to the `this` pointer. Concrete examples can be found in appendix D.

In order to identify a virtual base, we scan the constructor of the derived base for calls to other constructors, we then analyze the offsets added to the `this` pointer before the calls are made. If an offset equals any vbase-offset found in the derived class' primary VTable, we conclude that the constructor being called belongs to a virtual base of the derived class. Lastly, we retrieve the primary vptr corresponding to the identified virtual base and then record it as a virtual base vptr.

Recovering Intermediate Bases A special constructor is used to construct intermediate bases. This special constructor has three major distinctions from a regular constructor. It takes two default argument, `this` pointer and subVTT address (lines 7 and 13 of Listing 2), unlike the regular constructors whose only default argument is the `this` pointer. The subVTT address contains pointers to the construction VTable of the IntermediateBase-in-Derived. Second, it does not call the constructors of any of its virtual bases (they are called by the classes that derive from it). Third, it does not initialize its vptrs using immediate values, rather it accesses the subVTT it received as argument to get the vptrs needed for initialization.

The third distinction will make the well known method of identifying constructors and destructors to fail since vptr initialization is not done using immediate value. The calls on lines 10 and 16 of Listing 2 will be seen as just a function call. As a result, no relationship will be identified between B and D, and C and D. To address this problem, we make use of another information that the constructor of a derived class exposes. For every intermediate base, the derived class calls a special constructor which takes a subVTT address (an immediate value) as its second argument. Therefore, to recover intermediate bases, we scan constructors for subVTT addresses. All those addresses represent individual intermediate bases. Once we have all the subVTT addresses, we retrieve their corresponding VTable addresses from the map obtained in Section 6.1 and then record them as intermediate bases of the derived class.

Building Virtual Inheritance Tree After recovering classes involved in virtual inheritance, including their virtual and intermediate bases, we merge the results to construct the virtual inheritance tree. We also build the overall inheritance tree which includes single, multiple and virtual inheritance.

6.3 Support for MSVC ABI

MSVC ABI's implementation of structures used for virtual inheritance is slightly different from Itanium ABI's implementation. Unlike the Itanium ABI where vbase-offsets, offset-to-top, RTTI (if present) and virtual functions are together within the VTable, MSVC ABI separates offsets; vbase-offsets are placed in a table named the virtual base table (VB-Table) while RTTI and virtual functions are placed in the VTable.

We recover virtual inheritance from a binary that follows the MSVC ABI implementation by first recovering VTables, this process is the same for the Itanium ABI. Next, we recover VB-Tables (Algorithm 4). VB-Tables seem more difficult to precisely recover than VTables, however, we noticed that the first entry in every VB-Table we found is a particular constant value. We used that constant value as a signature to recover them. We recover constructors and destructor next, by looking for functions which initialize vptrs (same for Itanium ABI). The first VB-Table of a given type contains offset(s) to the virtual base(s) from the top of the object. Therefore to identify virtual bases (Algorithm 5), we look for calls in constructors and destructors whose first argument is the address of the object plus an offset contained in the first VB-Table of the object type (similar to Itanium ABI). VTTs are not present in MSVC ABI binaries, therefore, intermediate bases are recovered differently. Since intermediate bases have direct or indirect virtual bases, they initialize VB-Table ptrs. We recover intermediate bases by looking for calls to constructors which initialize known VB-Table ptrs (Algorithm 5). Note: MSVC ABI does not have construction VTables, however, the VB-Tables contain similar offset values as construction VTables. Therefore, virtual inheritance in a binary that adheres to MSVC ABI can also be exploited.

7 IMPLEMENTATION AND EVALUATION

We developed `VirtAnalyzer` as an IDA Python plugin that builds on top of DeClassifier, an existing class hierarchy inference engine that infers single and multiple inheritances. `VirtAnalyzer` can infer virtual inheritance from binaries that adhere to both Itanium and MSVC ABIs irrespective of the compiler used to compile the program (gcc, clang or visual studio). .

We aim to answer the following questions in our evaluation:

- How accurately can virtual inheritance tree be recovered from a stripped binary?
- How prevalent is the use of virtual inheritance in MSVC binaries?
- How does the presence of virtual inheritance reduce the effectiveness of state of the art binary level defenses like Marx?
- How accurately can the overall class inheritance (single, multiple and virtual) tree be recovered (Appendix E)?

All ELF binaries were compiled with GCC 7.3.0 (clang+llvm 7.0) on Ubuntu 18.04 LTS, whereas Windows PE binaries were compiled using Visual Studio on Windows 10 OS. All experiments were performed on Intel core i7 3.60GHz with 32GB RAM. We compared the results from our analysis with the ground truth. Ground truth (GT) for all the binaries except mysqld, mysqlbinlog and mysqlpump were obtained from GCC's compilation option -fdump-class-hierarchy, which dumps a representation of the hierarchy of each class, including their VTable layout [18] and VTTs. Mysqld, mysqlbinlog and mysqlpump are together in a single package, therefore, to know their distinct inheritance trees, we analyzed RTTI structures in their binaries.

7.1 VTT Recovery

We report the number of distinct VTTs recovered from binaries in Table 5. VTTs are reliable indication of virtual inheritance in a given binary. VTTs could also be used as a basis for comparing two binaries for similarity. Recovering a sufficient number of VTTs from two binaries and analyzing their entries can indicate if they are similar or not. Lastly, VTTs can be reliably used to verify VTables, and to differentiate them from construction VTables. As shown in the table, the number of VTTs recovered range from 1 to 166.

Table 5: Number of VTTs recovered.

Program	# of VTTs
libabw, boost_date_time, libcdr, libgdcmMEXD, libGLU, libopencv_phase_unwrapping, librevenge-generators, librevenge-stream	1
libgdcmDSED, libopencv_features2d, libopencv_structured_light, libphonenumber, VBoxRT	2
libepub-gen, libetonyek, librados, libglibmm-2.4	3
libsocket++	4
boost_iostream, boost_locale, libgdal	5
libcmis-c, libstorelo	6
boost_thread	8
libopencv_saliency	9
libstdc++	27
cmake, ctest, cpack, btag, k4dirstat, kgeography, scantailor	1
bedtools, between,	2
grfcodec, primrose	3
gpick, xboxdrv, mysqlbinlog	6
fityk	7
mysqlpump	10
x86_64-linux-gnu-ld.gold	11
x86_64-linux-gnu-dwp	12
darkice	22
ragel	45
Mysqld	166

7.2 Virtual Inheritance Recovery

Table 6 shows our analysis result for virtual inheritance compared with the ground truth. First, we identified classes which have at least one virtual base. After that, the number of direct bases and number of intermediate bases which those classes have was counted. It is possible to have a class with only virtual bases and no intermediate, that is the reason we have a column "0" under "Intermediate bases". We used ">1" to denote all other numbers of virtual and intermediate bases because most classes have 2 or less numbers of virtual and intermediate bases. The compiler may choose to eliminate an entire class from the compiled binary, for instance, because such class is not initialized in the program. We removed such classes from the GT used for comparison.

For libraries, we correctly recovered 95% of virtual bases and 95.5% of intermediate bases. We underestimated 5% of virtual bases and 4.5% of intermediate bases. For executables, we correctly recovered 98.8% of virtual bases and underestimated 1.2% of them, also, we correctly recovered 76% of intermediate bases and underestimated 24%. In DealII, “LaplaceSolver::PrimalSolver<3>" (LP) has “LaplaceSolver::Solver<3>" (LS) as its intermediate base, but the main constructor (or destructor) of LP where the VTT entry of LS should be initialized is not present in the binary. As a result we missed the relationship between LS and LP. Five of the six classes with underestimated intermediate bases have LP as intermediate base, while LP is the sixth class. This is the case for most of the underestimations. No overestimation was recorded.

7.3 Recovery for Higher Levels of Optimization

In order to ascertain that VirtAnalyzer performs effectively with higher levels of optimization, we compiled the library test set with the default compiler flags specified by the library authors in the configuration files. We did not alter the default configurations which we verified to be O2 optimization for all the libraries. This approach of evaluating with default compiler options is consistent with existing works [31, 35]. This evaluation is done by compiling with both GCC and Clang (Table 7). As expected, with higher levels of optimization, some constructs (e.g. VTables) needed for recovery are missing in this binary. This is not a limitation of our tool because the needed information is infact not in the binary. For GCC, we correctly recovered 83.3% and 82.9% of virtual and intermediate bases. We underestimated 16.7% and 17.1% while we overestimated 3.6% and 1.4% of virtual and intermediate bases respectively. For Clang binaries, we correctly recovered 77.5% and 73.8 of virtual and intermediate bases. We underestimated 22.5% and 26.2% while we overestimated 2.5% and 0% of virtual and intermediate bases respectively. We were unable to compile Libstdc++6 with clang, while the other 5 binaries(e.g libcmis-c) which have smaller numbers virtual inheritance occurrences do not contain any VTT when compiled with clang, default optimization.

7.4 MSVC Binaries

We analyzed 648 DLLs which we recovered from various directories (including Program Files) of a Windows machine. The aim of this evaluation is to find out how prevalent the use of virtual inheritance is among Windows applications. All these DLLs contain polymorphic classes. Of the 648 DLLs, we found 81 (12.5%) to contain instances of virtual inheritance. Table 8 shows the DLLs with the top 5 (for the reason of space) number of classes with virtual inheritance. We manually checked the binaries to verify that the reported numbers are true positives. Figure 6 shows the distribution of the number of classes with virtual inheritance among the 81 DLLs. The x-axis shows the number of classes while the y-axis shows the number of DLLs with those numbers.

7.5 Comparison with Marx

We compare VirtAnalyzer with Marx (Table 9), by considering the number of classes in virtual inheritance tree which Marx and VirtAnalyzer recovered. We attempted to do similar comparisons with VCI and SmartDec. However, VCI is not open sourced and the authors did not release the source code to us. We compiled a version of SmartDec that we found on GitHub, but the tool does not

Table 6: Table showing the # of classes in virtual inheritance tree. The “#Classes with virt inh" are classes with atleast 1 direct or indirect virtual base, the “0" subcolumn intermediate bases implies only direct bases , “#matching" represents the # of bases that were correctly recovered, “#over-est" and “#under-est" imply that we recovered more and less bases than the GT respectively and “#not found" are the virtual inheritance instances we missed. Under “Intermediate bases", some entries have “N/A", this is because those programs have no class with an intermediate base.

Program	#Classes with virt inh	Virtual Bases					Intermediate bases						#not found
		1	>1	#matching	#overest	#underest	0	1	>1	#matching	#overest	#underest	
libstdc++6	29	29	0	29	0	0	8	19	2	27	0	2	1
libcmis-c	6	6	0	6	0	0	6	0	0	N/A	N/A	N/A	5
libcmis	26	26	0	26	0	0	26	0	10	26	0	0	6
libcdr	2	2	0	2	0	0	2	0	0	N/A	N/A	N/A	0
libepub-gen	3	3	0	1	0	0	3	0	0	N/A	N/A	N/A	1
libetonyek	3	3	0	3	0	0	3	0	0	N/A	N/A	N/A	0
libgdal	15	15	0	13	0	2	1	14	0	14	0	1	3
librados	3	3	0	3	0	0	3	0	0	N/A	N/A	N/A	0
mysqld	201	191	10	201	0	0	65	86	50	200	0	1	0
mysqlbinlog	16	16	0	16	0	0	10	0	6	16	0	0	0
mysqlpump	26	26	0	26	0	0	15	1	10	26	0	0	0
DealII	8	8	0	8	0	0	4	3	1	8	0	6	1
Ragel	47	47	0	46	0	1	13	6	28	25	0	22	0
Darkice	14	4	10	12	0	2	3	10	1	5	0	9	8
Between	2	2	0	2	0	0	1	1	0	2	0	0	0
Btag	1	1	0	1	0	0	1	0	0	N/A	N/A	N/A	0
gpick	5	5	0	5	0	0	5	0	0	N/A	N/A	N/A	3
grfcodec	3	3	0	3	0	0	3	0	0	3	0	0	4
primrose	3	3	0	3	0	0	1	2	0	3	0	0	0
Scantailor	1	1	0	1	0	0	1	0	0	N/A	N/A	N/A	0
xboxdrv	5	5	0	5	0	0	5	0	0	N/A	N/A	N/A	1

Table 7: Library test set compiled with higher levels of optimization using both GCC and Clang

Compiler	Program	#Classes with virt inh	Virtual Bases					Intermediate bases						#not found
			1	>1	#matching	#overest	#underest	0	1	>1	#matching	#overest	#underest	
GCC	libstdc++6	26	26	0	26	0	0	4	17	5	25	0	0	4
	libcmis-c	6	6	0	6	0	0	6	0	0	N/A	N/A	N/A	5
	libcmis	20	17	3	17	3	0	11	1	8	19	1	0	12
	libcdr	1	1	0	1	0	0	1	0	0	N/A	N/A	N/A	1
	libepub-gen	3	3	0	1	0	0	3	0	0	N/A	N/A	N/A	1
	libetonyek	3	3	0	3	0	0	3	0	0	N/A	N/A	N/A	0
	libgdal	15	15	0	13	0	2	1	14	0	14	0	1	3
Clang	libcmis	17	17	0	17	0	0	7	0	10	17	0	0	15
	libgdal	16	14	2	14	2	0	1	15	0	14	0	2	2

Table 8: MSVC Binaries (Top 5 with virtual inheritance)

DLLs	Virtual Bases		
	#classes with virt inh	1	>1
migcore	382	292	90
igd11dxva32	67	2	65
igd9dxva32	65	5	60
igd12dxva32	65	5	60
igd9dxva64	64	37	27

do what the paper describes. It tries to recover source code from a binary rather than recover class hierarchy. Marx groups classes into sets while we assign direction of inheritance to every class. To achieve a fair comparison, we evaluated the number of distinct virtual inheritance trees which we recovered. Column "#cvtables" shows the number of construction VTables which Marx groups with regular VTables (these constitute false positives). For all the binaries, except Libstdc++ and Libetonyek, Marx groups classes involved in virtual inheritance into 1 or 2 sets. These sets contain the virtual bases, intermediate bases and other classes with either single or multiple inheritance. For Libstdc++, Marx groups each class with virtual inheritance into separate sets with no other class in them. None of the sets contain either virtual or intermediate bases, they were missed. For Libetonyek, Marx groups classes involved in virtual inheritance into 3 separate sets. Those sets also contain other classes not involved in virtual inheritance with neither the virtual or intermediate bases being present. Lastly, Marx does not reason about virtual inheritance, as a result column "#Edges in VIT" is zero for all binaries.

8 RELATED WORK

VCI analyzes constructors to recover single and multiple inheritance tree. It uses the order of constructor calls to identify the base classes of a derived class. It does not consider virtual inheritance. Marx [31] is slightly similar to VCI, however, it uses a more intuitive approach.

Table 9: Table comparing the representation of Marx for class involved in virtual inheritance with `VirtAnalyzer` for both libraries and executables. "#set" shows the number of sets recovered by Marx containing at least one class with a virtual base class. "#classes in set" shows the total number of classes in the sets. "#cvtables" is the total number of constructor VTables wrongly identified as regular VTables. "#Edges in VIT" shows number of edges in Virtual Inheritance Tree

Program	Marx				Our analysis			
	#set	# classes in set	#cvtables (falses)	#Edges in VIT	#tree	#in tree	#cvtables (falses)	#Edges in VIT
libstdc++	26	32	2	0	2	30	0	31
libcmis-c	1	7	0	0	1	7	0	6
libcmis	2	48	20	0	2	28	0	36
libcdr	1	3	0	0	1	3	0	2
libepub-gen	1	4	0	0	1	4	0	3
libetonyek	3	4	0	0	1	4	0	3
libgdal	1	31	15	0	1	16	0	15
librados	1	4	0	0	1	4	0	3
mysqld	8	271	69	0	6	202	0	259
mysqlbinlog	1	30	10	0	4	20	0	22
mysqlpump	3	41	15	0	2	26	0	59
DealII	2	18	10	0	2	10	0	9
ragel	1	100	50	0	3	50	0	70
Darkice	1	23	4	0	1	19	0	32
Between	1	3	0	0	1	3	0	2
gpick	1	6	0	0	1	6	0	5
grfcodec	1	4	0	0	1	4	0	3
primrose	1	4	0	0	1	4	0	3
xboxdrv	1	6	0	0	1	6	0	5

During calls to base class constructors or destructors, the vptr within the derived class object gets overwritten. Only vptrs of related classes can be overwritten. Marx analyzes vptr overwrites to group related classes. Its weakness is in its inability to differentiate between a constructor and destructor which makes it impossible to assign direction of inheritance. Marx also does not reason about virtual inheritance, it simply groups vptrs for construction VTables and complete object VTables together.

SmartDec [16] considers both constructor and destructors to recover inheritance. Like VCI, it identifies the base classes by considering the calls to constructors and destructors in the derived class' constructor and destructor respectively. It only considers single and multiple inheritance, not virtual inheritance.

OOAnalyzer [35] takes a different approach in recovering high level semantics in C++ programs. It does not rely on VTables which makes it possible to consider both polymorphic and non-polymorphic classes. However, this makes it hard to recover inheritance since vptr initialization is a strong indication of class relationships.

Katz et al.[26] proposed an approach to statically determine the possible targets of virtual function calls. This is achieved by identifying object tracelets. The object tracelets are used to train a statistical language model (SLM). Basically, the ensemble of SLMs is used to measure the likelihood that sets of tracelets share the same source, those set of tracelets are grouped together, which then form the basis for predicting possible targets of virtual function calls. The grouping of object types is similar to what Marx does.

DeClassifier [11] implements several techniques to recover class hierarchy information from optimized binaries. These techniques include constructor/destructor analysis, overwrite analysis and object layout analysis. It achieves a high precision and accuracy on optimized binaries, however, it does not recover virtual inheritance.

vfGuard [33] is a binary level defense that protects virtual function callsites. It statically analyzes the binary to recover VTables as well as function offsets specified at callsites. It uses this information to enforce a CFI policy that restricts virtual dispatch targets to only functions at the same offset within VTables as the static offset specified at callsite.

VTable Pointer Separation (VPS) [32] is a binary level defense that implements CFIXX's [30] Object Type Integrity on the binary level. It performs static and dynamic analysis and runtime instrumentation to identify the target vptr allowable at a given virtual callsite.

9 CONCLUSION

Previous binary level class hierarchy recovery solutions have made no attempt to recover the virtual inheritance tree of a program. In this work, we show the security implications of the failure to include virtual hierarchy in the overall inheritance tree. We also present simple, but efficient algorithms to recover virtual inheritance in a C++ binary that adheres to either Itanium or MSVC ABI.

10 ACKNOWLEDGEMENT

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

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*.

[3] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. Byteweight: Learning to recognize functions in binary code. In *USENIX Security Symposium*.

[4] 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)*.

[5] Yue Chen, Mustakimur Khandaker, and Zhi Wang. 2017. Pinpointing Vulnerabilities. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS'17)*.

[6] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M. Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling Execute-Only Memory for COTS Binaries on. In *2017 IEEE Symposium on Security and Privacy Proceedings (SP'17)*.

[7] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures from Binaries. In *Proceedings of the 26th USENIX Conference on Security Symposium*.

[8] David Dewey and Jonathon T. Giffin. 2012. Static detection of C++ vtable escape vulnerabilities in binary code.. In *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)*.

[9] 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*.

[10] Bauman Erick, Lin Zhiqiang, and W. Hamlen Kevin. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*.

[11] 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)*.

[12] Rukayat Ayomide Erinfolami, Anh T Quach, and Aravind Prakash. 2019. On Design Inference from Binaries Compiled using Modern C++ Defenses. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19)*.

[13] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: attacking branch predictors to bypass ASLR. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.

[14] Sui Fan, Liao Xiaokang, Xue Yulei, and Jingling Xiangke. 2017. 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)*.

[15] Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. 2016. ORIGEN: Automatic Extraction of Offset-Revealing Instructions for Cross-Version Memory Analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIACCS'16)*.

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

[17] 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)*.

[18] GCC. 2019. *GCC Developer Options*.

[19] 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)*.

[20] Istvan Haller, Yuseok Jeon, Peng Hui, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2016. TypeSan: Practical Type Confusion Detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[21] Niranjan Hasabnis and R. Sekar. 2016. Extracting Instruction Semantics via Symbolic Execution of Code Generators. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[22] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *2013 IEEE Symposium on Security and Privacy (SP'13)*.

[23] 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)*.

[24] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[25] 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)*.

[26] Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. Estimating Types in Binaries using Predictive Modeling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[27] 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)*.

[28] 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)*.

[29] Paul Muntean, Sebastian Wuerl, Jens Grossklags, and Claudia Eckert. 2018. CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM. In *23rd European Symposium on Research in Computer Security (ESORICS'18)*.

[30] 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)*.

[31] 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*.

[32] 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)*.

[33] 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)*.

[34] Pawel Sarbinowski, Vasileios P. Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: Practical VTable Hijacking Protection for Binaries. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications (ACSAC'16)*.

[35] 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*.

[36] Igor Skochinsky. 2011. Practical C++ decompilation. https://archive.org/details/Recon_2011_Practical_Cpp_decompilation

[37] 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)*.

[38] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proceedings of IEEE Symposium on Security and Privacy*.

[39] Srinivasan Venkatesh and Thomas Reps. 2014. Recovery of Class Hierarchies and CompositionRelationships from Machine Code. In *23rd International Conference on Compiler Construction (CC'14)*.

[40] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*.

[41] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*.

[42] 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)*.

[43] 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)*.

[44] Mingwei Zhang, Michalis Polychronakis, and R. Sekar. 2017. Protecting COTS Binaries from Disclosure-guided Code Reuse Attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC'17)*.

[45] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (Usenix Security'13)*.

A DEFINITION OF TERMS

Polymorphic class: A class that declares at least one virtual function or derives directly or indirectly from a class that is polymorphic.

Direct and Indirect base: Class DB is said to be the direct base of class C if C inherits directly from DB. Whereas, class IB is said to be an indirect base of class C if there exists at least one class M such that M inherits from IB and C inherits from M.

Primary and Secondary VTables: Primary VTable of a class C contains the virtual functions defined in C. It is shared with C's primary base. C has a secondary VTable associated with each of its secondary bases. The secondary VTables of a class are laid out immediately after the primary VTable, as a result, the address of a secondary VTable is always greater than that of its primary VTable.

Object and Sub-object: An object of class C contains entries for vptrs to C's VTables and entries for all non-static member variables of C. A sub-object in C's object belong specifically to C or one of its base classes and it contains a vptr and non-static member variables defined in C or the base class. For instance, Figure 3 shows that C's object contains two sub-objects.

Virtual Inheritance Tree: This a subtree in the class hierarchy tree rooted at a virtual base.

Virtual Call Offset Every virtual function defined in the virtual base class has a vcall-offset entry in the secondary VTable (of the derived) corresponding to the virtual base. Since the virtual base could be shared among multiple base classes of a derived class (e.g. B and C in the running example), there is the need to identify the derived class with the most recent definition. The associated vcall-offset is equal to the offset of the virtual base sub-object from the derived sub-object with the most recent definition. Functions which are not overriden by a derived class have vcall-offset of zero, while the others have negative vcall-offsets.

B ALGORITHMS

The algorithms used to recover virtual inheritance are given in Algorithms 1-5.

C DATA USED BY VIRTANALYZER

Table 10 shows data used in each sub-phase of VirtAnalyzer.

Table 10: Table showing steps in recovering virtual inheritance and the data involved in each step

Step	Data involved
Identifying VTables	VTables
Identifying VTTs	VTables, VTTs
Grouping subVTTs	VTTs, subVTTs
Extracting virtual base offsets	VTables, subVTTs
Mapping constructor VTables to regular VTables	Construction VTables, VTables, VTTs, subVTTs
Identifying constructors destructors	Ctors, Dtors, VTables
Parsing ctors and dtors	Ctors, Dtors
Recovering virtual bases	VTables, Ctors, Dtors, vbaseoffset
Recovering intermediate bases	subVTTs, Ctors, Dtors

Algorithm 1 IdentifyAVTT.

```
procedure IdentifyAVTT($addr$, $nextVTTIndex$)
    $vptr \leftarrow getvptrAtAddr(addr)$
    if $isInSegment(vptr, "data")$ then
        if $isAValidVTable(vptr)$ then
            $newVTT \leftarrow \emptyset$
            $newVTT.append(addr)$
            $nextEntry \leftarrow getNextVTTEntry(addr)$
            while $nextEntry! = -1$ do
                $nextVptr \leftarrow getvptrAtAddr(nextEntry)$
                if $nextVptr < vptr$ then
                    $break$
                end if
                if $isAValidVTable(nextVptr)$ then
                    $newVTT.append(nextEntry)$
                else
                    $break$
                end if
                $nextEntry \leftarrow getNextVTTEntry(nextEntry)$
            end while
            if $len(newVTT) > 1$ then
                $VTTs[nextVTTIndex] \leftarrow newVTT$
                $nextVTTIndex + +$
            end if
        end if
    end if
    return $nextVTTIndex$
end procedure
```

Algorithm 2 GroupSubVTTs.

```
procedure GroupSubVTTs(aVTT, VTables)
    $ordered\_vptr \leftarrow \emptyset$
    for each $addr$ in $aVTT$ do
        $ordered\_vptr.append(addr)$
    end for
    $ordered\_vptr.sort()$
    $k = -1$
    for each $addr$ in $ordered\_vptr$ do
        $ott \leftarrow VTables[addr]['offsetToTop']$
        if $ott == 0$ then
            $k = addr$
            $SubVTTs[k] \leftarrow \emptyset$
            $SubVTTs[k].append(k)$
        else
            if $k == -1$ then
                $continue$
            end if
            $SubVTTs[k].append(addr)$
        end if
    end for
end procedure
```

Algorithm 3 ExtractVBaseOffsets.

```
procedure ExtractVBaseOffsets(aSubVTT, VTables)
    $vptr \leftarrow getPryVptr(aSubVTT)$
    $curLoc \leftarrow vptr - (DWORD\_SIZE * 3)$
    $vBaseOffs[vptr] \leftarrow \emptyset$
    for each $i$ in $aSubVTT$ do
        $ott \leftarrow VTables[i]['offsetToTop']$
        $vbo \leftarrow Dword(curLoc)$
        if $ott == neg(vbo)$ then
            $vBaseOffs[vptr].append(vbo)$
            $curLoc \leftarrow curLoc - DWORD\_SIZE$
        end if
    end for
end procedure
```

Algorithm 4 GetVB_Tables.

```
procedure GetVB_Tables
    $imms \leftarrow getImmediatesFrmText()$
    for each $i$ in $imms$ do
        if $Dword(i)! = getVbtableConstant()$ then
            $continue$
        end if
        $entries \leftarrow \emptyset$
        $nLoc \leftarrow i + DWORDSIZE$
        while $True$ do
            if $Dword(nLoc) > 0\&\&Dword(nLoc) < CAPOFFSET$ then
                $entries.add(Dword(nLoc))$
                $nLoc \leftarrow nLoc + DWORDSIZE$
            else
                $break$
            end if
            if $len(entries) > 0$ then
                $vbtables[i] \leftarrow entries$
            end if
        end while
    end for
end procedure
```

Algorithm 5 RecoverVBasesAndIBases.

```
procedure RecoverVBases(ctorsDtors, vb_tables)
    for each $cd$ in $ctorsDtors$ do
        $vbptr \leftarrow getFirstVbaseptr(cd)$
        $vptr \leftarrow getAssVtable(cd)$
        $callInstns \leftarrow getCallInstns(cd)$
        for each $(addr, targ)$ in $callInstns$ do
            $addedOffset \leftarrow getAddedOffset(addr)$
            $vptr\_t \leftarrow getAssVtable(targ)$
            if $addedOff$ in $vb\_tables[vbptr]$ then
                $VirtualBases[vptr].add(vptr\_t)$
            end if
            if $initVbptr(targ)$ then
                $IntermBases[vptr].add(vptr\_t)$
            end if
        end for
    end for
end procedure
```

D EXAMPLE OF SHOWING VBASE-OFFSETS ADDED TO THE THIS POINTER

In Listing 8, lines 3, 7 and 11, offsets of 0x10, 0x20 and 0x30 respectively (vbase-offset of A-in-D,B-in-D and C-in-D) are added to the `this` pointer before the call instructions on lines 5, 9 and 12 respectively. Similarly, in Listing 2, line 3, offset of 0x20 (vbase-offset of A-in-D) is added to the `this` pointer.

E OVERALL CLASS INHERITANCE RECOVERY

Table 11 shows our result for the overall class inheritance recovery compared with the ground truth. The table shows the number of classes, most base classes (classes with no base), classes with single base, classes with multiple bases and classes in virtual inheritance tree for GT and `VirtAnalyzer`. We recorded an average precision of 99.8% and 99.5% overall hierarchy for libraries and executables respectively. For recall, we recovered an average of 66.5% and 86.5% for libraries and executables respectively. Most of the missing classes come from the Boost libraries.

Listing 8: Disassembly of the ctor of D in Figure 1

```
...
1.  mov [rbp+var_8], rdi
2.  mov rax, [rbp+var_8]
3.  add rax, 10h
4.  mov rdi, rax; this, at offset 10h
5.  call _ZN1AC2Ev; A::A(void)
...
6.  mov rax, [rbp+var_8]
7.  add rax, 20h
8.  mov rdi, rax; this, at offset 20h
9.  call _ZN1BC2Ev; B::B(void)
...
10. mov rax, [rbp+var_8]
11. add rax, 30h
11. mov rdi, rax; this, at offset 0
12. call _ZN1CC2Ev; C::C(void)
...
```

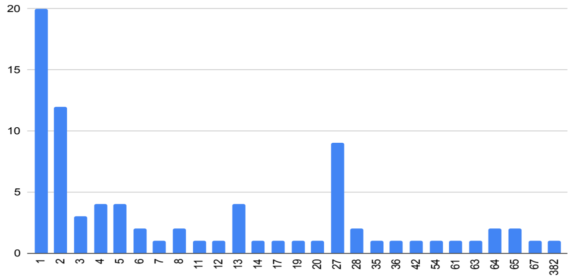


Figure 6: MSVC: # of classes with virtual inheritance

F DISTRIBUTION OF NUMBER OF CLASSES WITH VIRTUAL INHERITANCE IN MSVC BINARIES

Figure 6 shows the distribution of the number of classes with virtual inheritance among the 81 DLLs which were analyzed. The x-axis shows the number of classes while the y-axis shows the number of DLLs with those number of classes.

Table 11: Table showing the overall class hierarchy from the analysis compared to the ground truth for both libraries and executables

Program	#Classes		#most base classes		# with single base		#with multiple bases		#with virt inh	
	GT	Analysis	GT	Analysis	GT	Analysis	GT	Analysis	GT	Analysis
libstdc++	211	202	11	12	172	164	1	0	27	26
libcmis-c	78	30	39	11	21	10	7	3	11	6
libcmis	230	194	54	28	134	133	13	7	32	26
libcdr	181	57	48	20	69	33	6	2	58	2
libepub-gen	90	77	27	15	59	59	2	2	2	3
libetonyek	1008	814	50	42	884	768	17	1	57	3
libgdal	1103	1024	131	146	948	855	6	8	18	15
librados	213	120	86	56	240	58	13	3	12	3
mysqld	3640	3666	252	408	3144	3010	46	47	198	201
mysqlbinlog	66	71	5	15	30	24	15	16	16	16
mysqlpump	117	123	10	19	77	74	4	4	26	26
DealII	874	711	25	22	836	678	4	3	9	8
ragel	74	73	3	2	24	24	0	0	47	47
Darkice	32	29	10	13	0	0	0	0	22	16
Between	47	45	15	16	27	24	3	3	2	2
gpick	82	63	25	16	43	38	6	4	8	5
grfcodec	45	30	18	12	11	16	7	4	7	3
primrose	28	27	7	7	18	17	0	0	3	3
xboxdrv	156	150	22	23	118	119	6	4	6	5