

Static detection of C++ vtable escape vulnerabilities in binary code

David Dewey Jonathon Giffin

School of Computer Science, Georgia Institute of Technology
{ddewey, giffin}@gatech.edu

Abstract

Static binary code analysis is a longstanding technique used to find security defects in deployed proprietary software. The complexities of binary code compiled from object-oriented source languages (e.g. C++) has limited the utility of binary analysis to basic applications using simpler coding constructs, so vulnerabilities in object-oriented code remain undetected. In this paper, we present vtable escape bugs—a class of type confusion errors specific to C++ code present in real, deployed software including Adobe Reader, Microsoft Office, and the Windows subsystem DLLs. We developed automated binary code analyses able to statically detect vtable escape bugs by reconstructing high-level objects and analyzing the safety of their use. We implemented our analysis in our own general object code decompilation framework to demonstrate that classes of object-oriented vulnerabilities can be uncovered from compiled binaries. We successfully found vtable escape bugs in a collection of test samples that mimic publicly disclosed vulnerabilities in Adobe Reader and Microsoft Excel. With these new analyses, security analysts gain the ability to find common flaws introduced by applications compiled from C++.

1. Introduction

Many security vulnerabilities in software arise due to violations of memory safety. Safety violations in object-oriented programs include type confusion [21], a vulnerability that occurs when a pointer points to an object of an incompatible type. By design, C++ permits the introduction of type confusion errors via its `static_cast` operator: pointers can be unsafely downcast to incompatible child types or cast through void. When vulnerable software subsequently accesses virtual object methods through incompatible pointers, *vtable escapes* may occur: the process interprets arbitrary memory beyond the bounds of a C++ object's vtable as a code pointer and unsafely transfers execution. Professional security analysts have identified vtable

escape vulnerabilities in widely deployed software, including Microsoft Excel [24], Adobe Reader [1], and Microsoft Windows subsystem DLLs [23].

Static detection of possible memory safety violations employs offline analysis of source or binary code to find unsafe execution paths in the software. Many common computing environments use proprietary software, so analysis requires reverse engineering of binary code. For example, an intrusion prevention system (IPS) provider needs to be able to develop a signature that detects attacks against a known vulnerability. An anti-virus vendor may need to integrate with an operating system in ways that are unpublished. The complexity of binary code analysis is driven by the architectural decisions, choice of source language, and compiler options selected by the software's developers. In this paper, we statically detect vtable escape vulnerabilities by addressing the challenges faced by static binary analysis of executable code that was originally developed in C++.

Object code compiled from C++ includes complexities resulting from the object-oriented nature of the language. When straight C code is compiled to its binary equivalent, the programmatic structure of the source code is largely left intact. Most C-level constructs translate directly into assembly code. In contrast, C++ constructs are lost as the compiler translates object-oriented source code into untyped assembly, producing binary code with widespread programmatic flow through dynamically-computed indirect calls. The result is compiled code that obscures vulnerabilities from static analyzers.

We address the complexity of reverse engineering proprietary software written for Microsoft Windows. Widely-used applications are written in C++ so that they can take advantage of the interoperability provided by the Windows API. Reverse engineers working with Windows software will regularly encounter C++ code, often more frequently than pure C. We create analysis passes in the IDA Pro disassembler and LLVM compiler framework that identify object instantiation and construction, compute vtable bounds, track flows of object pointers to vtable dispatch locations, and verify the safety of the vtable accesses. Our analyzer re-

verses x86 binary code compiled from C++ into a static single assignment (SSA) intermediate representation suitable for extensive analysis. We remove indirect control flows arising due to dynamic dispatch through vtables and reconstruct the call graph of the program. Subsequently, traditional static analysis algorithms applied to software written with simpler constructs once again produce results. We choose to use static data-flow analysis to detect the presence of vtable escape vulnerabilities in binary software that builds without compiler errors or warnings, helping analysts and developers catch defects unintentionally introduced into object-oriented software.

Our long-term goal is to fully automate the reversal and analysis of large production software like Excel and Reader. Towards this goal, we validate the effectiveness of our specific object reconstruction algorithms on targeted microbenchmarks that replicate the vulnerabilities found in commercial binaries. Extending the analysis from test samples to production-grade binaries requires additional engineering of x86 instruction reversal to account for the diversity of instructions present in real software. This engineering work remains in progress.

In summary, we make the following contributions:

- **Resolve vtable dispatch calls in compiled binaries:** We use a set of data flow analyses to programmatically resolve C++ virtual function calls as they exist in compiled code. We then generate a static call graph (or multiple versions of a static call graph) to enable existing analyses to operate on code employing dynamic dispatch.
- **Programmatically identify vtable escape vulnerabilities introduced by C++ developers:** We demonstrate that our virtual function resolution analysis can immediately identify a type-safety issue commonly introduced into enterprise-class closed source software.
- **Construct a general C++ decompilation framework:** We create a framework for reversing C++ compiled code into the intermediate representation used by the LLVM compiler infrastructure, allowing an analyst to employ any of the dozens of pre-built analyses that ship with LLVM. We implemented this system as a plugin for the popular IDA Pro disassembler.

2. Related Work

This paper spans several areas of research regarding static code analysis. We will cover topics in the areas of binary decompilation, binary data structure recovery, vulnerability detection systems, and existing static compiler analyses.

2.1. Binary Decompilation

Binary decompilation has been studied thoroughly in both academia and industry. Most binary decompilation is accomplished through the elevation of the binary code to some form of intermediate representation. Cousot and Cousot showed that by restructuring a language into an abstract representation, complex analyses are more easily implemented [8]. Following this concept, Song et al. developed the BitBlaze framework for binary decompilation and analysis [31]. Several open source and commercial tools exist specifically for the decompilation of binary code. For example, the popular Hex-Rays plugin for IDA Pro reverses binary code to a C-like intermediate representation [17]. Dullien and Porst developed the Reverse Engineering Intermediate Language (REIL) for their commercial product, Bindiff [13]. These tools all struggle with C++ compiled code, returning code fragments that are largely useless to analysts. Our work specifically tackles the problem of decompiling and analyzing binaries compiled from C++.

2.2. Binary Data Structure Recovery

One of the major tasks accomplished in this work is the ability to reconstruct an object by analyzing its representation in a compiled binary. Binary data structure recovery has been studied for use in host-based intrusion prevention systems, forensic analysis, and reverse engineering. For example, Dolan-Gavitt et al. [12] developed a dynamic-analysis system that creates attack detection signatures by monitoring kernel data structures in a way that is resistant to evasion. Similarly, Cozzie et al. developed Laika [9], a system that uses Bayesian unsupervised learning to detect the presence of data structures in memory indicative of a bot infection. Slowinska et al. [30] created a system that recovers data structures from a compiled binary for the purpose of reverse engineering.

While research has been done in the area of C++ object reconstruction [28], that work relies on access to the source code or runtime type information (RTTI). In this work, we reconstruct C++ objects from their compiled binary equivalent without access to any additional information. Intuitively, one can see how this is very similar to the type inference and data structure recovery systems mentioned above. We extend these concepts to allow for the identification of unsafe type-casting and other intricacies specific to C++.

2.3. Vulnerability Detection Systems

Software vulnerability detection systems have existed for many years. Lint, created in 1977, has the ability to find flaws in the source code of C programs [18]. Lint-like

systems have been developed over the years for the analysis of C programs. For example, Sparse [32] is a tool designed to find flaws in the Linux kernel. Splint [33] is the modern-day and maintained version of Lint, and Clang [7] is a popular compiler with built-in static analysis capability. Larochelle and Evans built upon these early works to statically detect the presence of buffer overflow vulnerabilities in source code [15, 19]. Shankar et al. developed a system to statically detect format string vulnerabilities [29]. ARCHER, a system developed by Xie, Chou, and Engler, uses a constraint solver to determine the safety of array accesses [37]. Similarly, Austin et al. created a system to detect pointer and array access errors [2]. Yet even with all the work in this area, Heelan points out that these problems are still unsolved [16].

As simple buffer overflows and format string vulnerabilities became increasingly rare, research focused on the detection of dynamic memory errors. Evans [14] and Bush et al. [5] present differing approaches to the detection of dynamic memory errors. These concepts were readily extended to the analysis of compiled C programs. Bugscam [4] is one of the oldest of these types of binary scanning tools and has been used to discover hundreds of vulnerabilities since its release. An entire industry has grown from these early tools: companies like Ounce Labs (now part of IBM), Coverity, Fortify Software (now part of HP), and Veracode all offer commercial products and services for the analysis of source code and compiled binaries. Most automated vulnerability detection falls apart in the face of C++. Viega et al. developed ITS4 [35]—a vulnerability scanner with support for C++, but at the source code level. In the work presented in this paper, we are able to perform all of our analyses without access to any source or other type information and focus specifically on vulnerabilities present in C++ compiled code.

2.4. Compile-Time Analyses

Many static analyses have been integrated directly into popular compilers to provide developers with warnings and errors as they are building their software. C++ analyses typically come in the form of type-checking and checks for const-ness and volatility and are fully enumerated in the C++ standard [27]. Significant research has attempted to extend required checks with virtual function call resolution in C++ programs. Bacon and Sweeny [3], for example, developed a static analysis algorithm to determine whether dynamic dispatch is truly necessary for a given method call. In cases where it is not, it can be replaced with a static function call, thus reducing the size of the compiled binary and the complexity of the program. Pande and Ryder [25, 26] and Calder and Grunwald [6] continue this concept to eliminate late binding where possible to take advantage of instruction

pipelining on modern-day processors. SAFECode, a system developed by Dhurjati, Kowshik, and Adve [11] introduces a new type system that can be enforced at compile-time to prevent several types of vulnerabilities.

These prior works demonstrate the different needs for virtual function call resolution, but they all share one common trait: they require access to the source code. Many security analysis needs exist in environments where the analyst will not have access to the source code, yet will still need to accomplish the tasks described above.

3. Background

The C++ additions to C create new ways for developers to introduce software vulnerabilities into their code. Some of these extensions introduce safety conditions, including the type confusion error leading to vtable escape vulnerabilities, that compilers cannot identify during code generation. In this section, we discuss these issues, the complexities that C++ introduces into reverse engineering process, and the assumptions that underlie our analyses of Section 4.

3.1. Silent Type Confusion

A number of C++ code-level defects do not present the developer with any sort of compile-time warning. For example, the `static_cast` operator (a) converts a pointer to a base class into a pointer to a derived class, or (b) converts a pointer to or from a void pointer. *There is no check for object congruence*; this is not a language error but an deliberate design choice to allow developers to insert unsafe casts into their software [34]. Casts through void pointers clearly deactivate all compiler type-checking for the pointer, but common developer documentation of `static_cast` omits this behavior, presenting only operation (a) [22, 36]. Both cast types (a) and (b) violate type safety, and uses of `static_cast` without additional safety checks by the developer can result in type confusion. Neither Microsoft Visual Studio nor g++ warn of unsafe static casts because to do so would violate the very purpose of the operator.

Consider the example code shown in Figure 1a. A human analyst can see that the method `debug` should never be called on an object of type `class1`. The `static_cast` operation deliberately permits this type of error in software, and both Visual Studio and g++ build the code without warning or error. Running the compiled code will crash, perhaps in an exploitable way. At a low-level, when this code executes, it attempts to dereference the fourth entry in the vtable for `class1`. `Class1`, however, only has two entries in its vtable causing this dereference to read arbitrary memory—a vtable escape bug.

This class of vulnerability has impacted widely deployed proprietary software. For example, in March 2010, Mi-

```

class class1 {
public:
    class1();
    ~class1();
    virtual void addRef();
    virtual void print();
};

class class2 : public class1 {
public:
    class2();
    ~class2();
    virtual void voidFunc1() {};
    virtual void debug();
};

int _tmain(int argc, _TCHAR* argv[])
{
    class1 C1;

    C1.addRef();
    C1.print();

    static_cast<class2*>(&C1)->debug();

    return 0;
}

```

(a) Excerpt of original source (member function implementations omitted)

```

.text:00401000 _wmain      proc near
.text:00401000
.text:00401000 var_20      = dword ptr -20h
.text:00401000 var_1C      = dword ptr -1Ch
.text:00401000 var_18      = dword ptr -18h
.text:00401000 var_14      = dword ptr -14h
.text:00401000 var_C       = dword ptr -0Ch
.text:00401000 var_4       = dword ptr -4
.text:00401000
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      and     esp, 0FFFFFFF8h
.text:00401006      push    0FFFFFFFh
.text:00401008      push    offset loc_401950
.text:0040100D      mov     eax, large fs:0
.text:00401013      push    eax
.text:00401014      sub     esp, 18h
.text:00401017      push    esi
.text:00401018      mov     eax, dword_403018
.text:0040101D      xor     eax, esp
.text:0040101F      push    eax
.text:00401020      lea    eax, [esp+2Ch+var.C]
.text:00401024      mov     large fs:0, eax
.text:0040102A      mov     [esp+2Ch+var.18], offset off_402138
.text:00401032      xor     eax, eax
.text:00401034      mov     [esp+2Ch+var.4], eax
.text:00401038      mov     [esp+2Ch+var.20], offset off_40214C
.text:00401040      mov     [esp+2Ch+var.1C], eax
.text:00401044      mov     byte ptr [esp+2Ch+var.4], 1
.text:00401049      mov     esi, ds:printf
.text:0040104F      push    offset Format ; "I'm in class1\n"
.text:00401054      mov     [esp+30h+var.14], 1
.text:0040105C      call   esi ; printf
.text:0040106A      mov     eax, [esp+34h+var.20]
.text:0040106E      mov     edx, [eax+0Ch]
.text:00401071      add     esp, 8
.text:00401074      lea    ecx, [esp+2Ch+var.20]
.text:00401078      call   edx
.text:0040107A      xor     eax, eax
.text:0040107C      mov     ecx, [esp+2Ch+var.C]
.text:00401080      mov     large fs:0, ecx
.text:00401087      pop     ecx
.text:00401088      pop     esi
.text:00401089      mov     esp, ebp
.text:0040108B      pop     ebp
.text:0040108C      ret.n
.text:0040108C _wmain      endp

```

(b) Compiled Binary

Figure 1: C++ code with a type-safety violation.

Microsoft patched a vulnerability in Excel that was the result of C++ object type confusion [24]. In April 2011, Adobe announced a 0-day type confusion vulnerability in their Flash Player [1] after exploits appeared in the wild. Another actively exploited type confusion vulnerability occurred in the Microsoft ATL [23], a set of C++ template code that ships with Visual Studio. Developers were inadvertently including the vulnerable code in their own projects.

3.2. Reverse Engineering C++ Software

When C++ code is compiled more high-level information is lost than with what is experienced in C, leading to many unsolved problems in C++ reverse engineering. Dynamic dispatch is one of the most significant challenges: C++ developers can optionally create objects in such a way that the methods of an object are called through indirection.

In object-oriented design, polymorphism can be achieved by constructing objects that implement dynamic dispatch. This allows for the substitution of a method's im-

plementation using the same interface. In C++, this is accomplished by declaring a member function `virtual`. All of the virtual functions have a corresponding pointer to the function's implementation in the vtable of the object. They are each stored in the order that they are declared in the object and referenced as an offset from the base of the vtable. For example, consider the code shown in Figure 1a. In this case, there are four virtual member functions in `class2`. When this code is compiled, these four functions appear in the vtable as shown in Figure 2b, and as calls are made to those member functions, they will appear as an *indirect call* to the base of the vtable plus the offset corresponding to the correct member function.

3.3. Assumptions

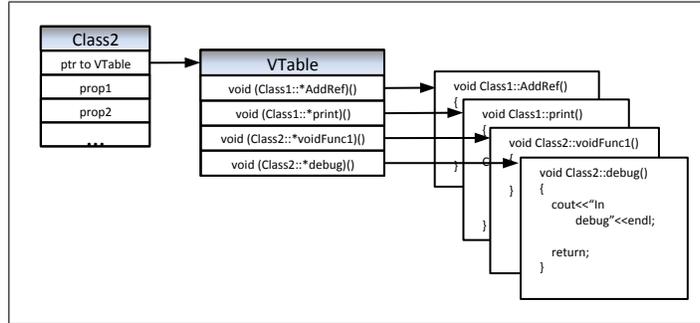
Our analyses and implementation provide automated tools that reduce the manual labor a reverse engineer must employ to better understand commercially available software. This may be needed to evaluate the security of a

```

.rdata:00402138 off_402138 dd offset sub_4010D0
.rdata:0040213C dd offset sub_4010A0
.rdata:00402140 dd offset nullsub.1
.rdata:00402144 dd offset sub_4010B0
.rdata:00402148 dd offset dword_402274
.rdata:0040214C off_40214C dd offset sub_4010D0
.rdata:00402150 dd offset sub_4010A0
.rdata:00402154 align 8
.rdata:00402158 db 48h ; H
.rdata:00402159 db 0
.rdata:0040215A db 0
.rdata:0040215B db 0
.rdata:0040215C db 0

```

(a) Disassembly of the vtables for class2 and class1



(b) Structure of a C++ object after compilation

Figure 2: Compiled objects in binary code

given program or to try to inter-operate with closed source software (e.g., creating a Microsoft Word rendering engine). The techniques presented in this paper are designed to analyze legitimate applications like these examples, and they may not have applicability to malware or other obfuscated programs. From a security perspective, legitimate, closed-source applications often require similar levels of analysis as required for malware because attackers will very often leverage software vulnerabilities in legitimate applications as a way to deliver malware payloads. Hence, understanding these vulnerabilities is also extremely important.

The binaries that we analyze throughout the paper were all compiled using Microsoft Visual Studio. We chose this as the target compiler for our analysis because most Windows programs that require reverse engineering are built in this environment. In contrast, applications that are built with the GNU developer tools are also usually open source and do not require the complex binary analyses we present here. However, all of the compiler-based issues we discuss are also present in g++, and our analyses could be extended to other platforms. The basis for the issues we discuss are rooted in the C++ standard rather than the implementation of that standard by the various compilers.

The analyses presented in this paper do not make use of any sort of Runtime Type Information (RTTI). As RTTI is only optionally compiled into the binary, we wanted to ensure our analysis would work in its absence.

4. Object Reaching Definition Analysis

As C++ code is compiled, it introduces constructs into the binary that make reverse engineering difficult. Notably, object methods declared as `virtual` introduce layers of indirection that can be nearly impossible to manually traverse. To assist in the analysis of C++ compiled code, we have created a data-flow analysis called Object Reaching Definition Analysis, with three goals:

1. Resolve the indirect virtual function calls present in binary code due to virtual function dispatch.
2. Statically construct the program's call graph to improve existing automated analyses.
3. Identify object vtable congruence failures.

Automated virtual function resolution allows analysts to easily navigate control flows in the compiled code. In resolving the virtual function calls, the static call graph becomes more accurate and enables existing analyses designed for C software to additionally operate on C++ software. Congruence failures at a point of virtual function dispatch indicate the presence of vtable escape vulnerabilities in the application.

Object Reaching Definition Analysis proceeds in five steps, each detailed in the remainder of this section. (1) Object identification locates instantiations of C++ aggregate objects in a binary program under analysis. (2) Constructor analysis determines the details, notably the vtable bounds, of each object. (3) Reaching definition analysis applies a standard fixed-point data-flow algorithm to determine which objects flow to which virtual function dispatch locations. (4) Virtual function resolution computes the vtable entry indexed at a dispatch site. (5) Object congruence evaluates the safety of each vtable access and warns of vtable bounds violations.

4.1. Object Identification

In the data flow analyses that will be introduced in the following subsections, we will need to be able to identify all of the new objects that are instantiated in a given basic block. When working directly with source code, identifying the instantiation of objects is rather easy. They are either declared on the stack as with any other sort of variable, or they are instantiated on the heap using the `new` operator. When analyzing binary code, we do not have access to these

obvious identifiers and must find other ways to locate the instantiation of new objects. Here we present four heuristics to detect the four ways an object instantiation can appear in binary code: either stack or heap creation, and either inlined or called constructors.

First, consider stack declaration with an inlined constructor (Figure 1a). Object `C1` is declared on the stack and Visual Studio is set to aggressively inline functions, so the constructors appear as shown in Figure 1b. We make the assumption that if we encounter a structure where the first element is a pointer to an array of function pointers that we are dealing with an object. In Figure 1b, the first element of a structure is being initialized to a pointer to an array of function pointers at `0x02A`. We now assume that this is an object and we will track this unique type throughout the rest of the program.

The second object instantiation we handle is when the object is declared on the heap using `new` and the constructor is inlined. In this case, we can be more precise in our determination that we are dealing with an object and not just a generic structure. We gain this precision because Visual Studio always applies the mangled name `YAPAXI` to the `new` operator, and `g++` similarly always applies the mangled name `Znwjxxx` (where `xxx` may vary) to `new`. We can apply the same logic as above, but only in cases where the pointer to an array of function pointers is being assigned to the value returned by one of the known `new` operators.

The third and fourth types of object instantiation occur when the objects are declared on the stack or heap, and the constructors are not inlined. In this case, we must employ an inter-procedural analysis to determine that we are dealing with an object. For this analysis, we make another assumption. Compilers will nearly always make the call to an object’s constructor immediately after the call to `new`. With that, we assume that the next call we encounter after a call to `new` is a constructor, and we can validate that assumption with the heuristics mentioned regarding the pointer to a table of function pointers.

4.2. Constructor Analysis

In the previous section, we covered the heuristics that we use to identify a constructor. To properly track an object’s use throughout the control flow of a program, we need to know more details about that object. Specifically, we need to know the size of the vtable, which function pointers the vtable stores, and the number and size of the properties stored within the object. These details can be extracted from the constructor.

To gain a full understanding of the vtable, we follow the pointer that is assigned to the first element of the object’s structure. Figure 2a shows the vtable for an object of type `class2` followed by the vtable for `class1` as referenced

in Figure 1a. We can see in the code in Figure 1b that the vtable starting at `0x138` is being referenced by the constructor for `class2`. In order to perform our later analyses, we need to understand the precise size of the vtable and the function pointers contained in that table. To determine the length of the vtable, we use three tests. First, if a data element is pointed to by another program point, it is likely the start of some other data structure (i.e. the next element past the end of our vtable). The element at `0x14C` is an example of this case. The second test is that if a pointer in the table does not point to a function, we assume that it is the next element past the end of the table. The element at offset `0x148` is an example of this case. The third hint is zero padding, as compilers will often pad the end of a data structure with zeros. Beginning at offset `0x154`, we can see zero padding past the vtable for `class1`.

4.3. Reaching Definition Analysis

Now that all of the object instantiations have been identified, we can analyze object flow to indirect call sites. We perform a fixed-point interprocedural reaching definition analysis for each object definition we encountered during the previous step. The data-flow equations used in reaching definition analysis are defined below.

For each basic block S :

$$REACH_{IN}[S] = \bigcup_{p \in pred[S]} REACH_{OUT}[p] \quad (1)$$

$$REACH_{OUT}[S] = GEN[S] \cup (REACH_{IN}[S] - KILL[S]) \quad (2)$$

where `GEN` is the set of objects that were identified as being instantiated using the heuristics listed in Section 4.2 and `KILL` is the set of objects that are deleted. In our analysis, objects must be tracked interprocedurally. In those cases, $REACH_{IN}$ at the entry of a function F is equal to $REACH[c]$ at the call site to F from a call site c .

4.4. Virtual Function Resolution

Given the set of objects that reach a given program point, we can resolve the virtual function calls that appear in the binary. There are several cases that can occur when evaluating the reaching definitions. In the first case, only a single object definition reaches. In the second case, a single object definition or `NULL` reaches. In the third case, a decidable number of object definitions reach. In the final case, an undecidable number of object definitions reach. We will discuss each of these cases below:

1. *Single Object Definition*

In this case, we can make the safe assumption that only a single object definition reaches a given program point. With this assumption, we can resolve the function calls made to methods of the object by simply indexing into the vtable based on the offset from the base. For example, in the code in Figure 1a, only a single object definition reaches. In the compiled equivalent in Figure 1b, we can see at line 0x078 there is a call through `edx`. Since we have reconstructed the vtable for the object, we can tell that it is actually a call to `debug`. As we resolve the virtual function calls, we can also check for congruence, as explained in Section 4.5.

2. *Single Object Definition with NULL*

Visual Studio sometimes adds a check for the return from the `new` operator and sets it to `NULL` in the case of failure. In these instances, it is possible for a developer to end up with a `NULL` pointer dereference even though they did not explicitly add this code. The code in Figure 3b at line 0x025 is an example of this check. In these cases, our analysis treats the object definition exactly as in Case 1. Further work can be done to determine the safety of this check insertion.

3. *Decidable Number of Object Definitions*

In this case, multiple object definitions reach a given program point, and we are able to determine the exact number and type of those objects. For each object definition, we perform the analysis to determine the safety of the use of the object as done in Case 1.

4. *Undecidable Number of Object Definitions*

This case occurs when objects are instantiated and stored in a manner that does not allow our heuristics to determine their type. This scenario is typically encountered when a class pointer is stored in a collection of some type. When a class pointer is stored and retrieved from the heap in a manner other than direct variable assignment, our reaching definition analysis will fail. An example of this would be instantiating an object and storing its pointer in a `std::map`. Since this object can now be referenced by the key value, we cannot check for congruence or resolve the virtual function calls.

4.5. Object Congruence

In addition to using object reaching definition analysis for virtual function resolution, we also test for *object congruence*. We define two objects as congruent when they are made up of the same number of methods and properties.

The method at each equivalent location in the two vtables must require the same number and type of arguments. The properties of the two objects must correspond in size and type at each offset in the property table.

As binary code is analyzed in the manner described in Section 4.4, when an object use is identified, we implement a congruence check to make sure the object is being referenced in a way that is safe with regard to the actual object type that reaches that use. This becomes particularly important when more than one object definition reaches a given use. We have to ensure that any object, when referenced, will be referenced safely. If even one of the possible reaching object definitions is unsafe for use, then the program point as a whole has to be marked unsafe because static analysis cannot guarantee which object definition will be actually used at runtime.

5. Implementation

In order to test and verify the data flow algorithms presented in Section 4, we created a framework called `RECALL`. This tool allows us to reverse x86 compiled binaries into the intermediate representation used by the LLVM compiler framework [20]. Then, we use the analysis capabilities in LLVM to implement the algorithms for Object Reaching Definition Analysis, virtual function resolution, and object congruence testing. This section presents the details of how we built this system.

5.1. High-Level Architecture

As compiled binaries are translated into the LLVM IR and analyzed, the data traverses several tools and formats. The diagram provided in Figure 4 shows the process described below.

The initial input into `RECALL` is x86 machine code compiled from x86 source. There are several commercially available and free tools that disassemble machine code into its assembly equivalent. `RECALL` uses `IDA Pro`, a commercially available disassembler with an extensive set of tools available for reverse engineers. It focuses almost entirely on analyzing the assembly representation of the compiled code. `IDA Pro` offers a plugin infrastructure whereby a developer can inter-operate with the analysis framework. In `RECALL`, we created a plugin for `IDA Pro` called `llvmbcwriter` that traverses the assembly code and translates it into the LLVM intermediate representation. The converted IR is then written to LLVM's bitcode format to be consumed by their analysis framework in the next step. This conversion process is described in detail in Section 5.2.

With a completed bitcode file, we can now implement the analyses detailed in Section 4 using the tool suite provided with LLVM. We use a tool called `opt` that allows de-

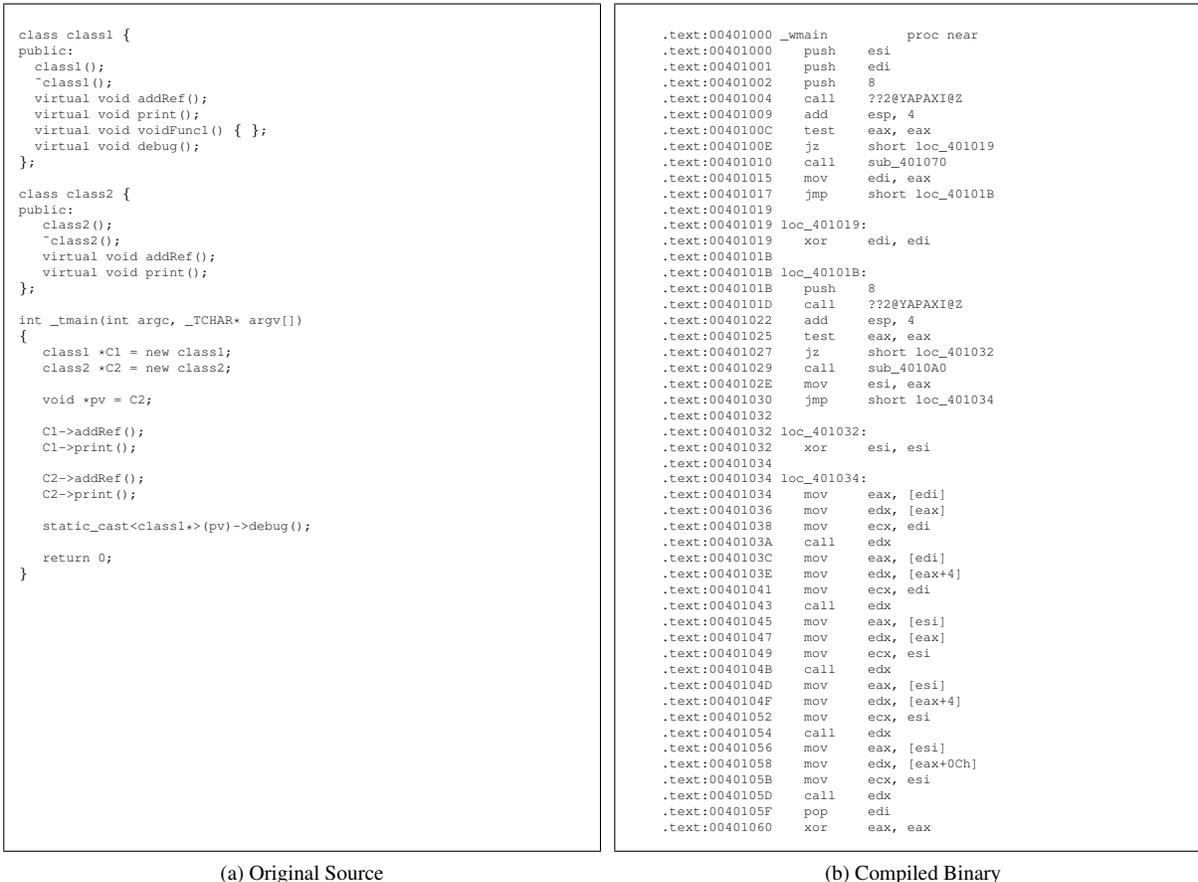


Figure 3: C++ code with heap-declared objects

velopers to test custom code analyses. It provides the same infrastructure that would be available inside the compiler without requiring compiled output. `opt` disassembles the bitcode into the LLVM IR and provides interfaces for analysis. The LLVM IR is a static single assignment (SSA) form, providing benefits to static analysis. For example, in SSA form the use-def chains are implicit, which greatly assists in the reaching definition analysis as described in Section 5.3.

Much like IDA Pro, `opt` provides a plugin infrastructure for custom analysis. In this phase of our framework, we created a plugin for `opt` called `ClassTracker` that performs a reaching definition analysis on objects as they appear in the LLVM IR. With this reaching definition analysis completed, we can resolve virtual function calls and perform type-safety checks on object usage. These procedures are covered in Section 5.3.

With the virtual function resolution performed by `ClassTracker`, we can now propagate that information back into the LLVM IR or even the disassembly. With this information provided in these lower-level formats, other analyses are now possible that were previously broken by dynamic

dispatch.

5.2. Decompilation

The first step of RECALL is to reverse the x86 machine code into the LLVM intermediate representation. `Llvm-bcwriter`, our IDA Pro plugin, makes several passes over the x86 assembly code to generate the IR. First, we create a generic class object that can later be referenced as instantiations of new object are identified. This is done early because in the LLVM framework, types must be predefined. The second step is to collect all the functions in the module. We identify their arguments, local variables, and return types, and create each function in the IR. Third, we insert each basic block into the functions we have just created. Then, for each basic block, we insert the individual instructions. Additional low-level detail of the decompilation process can be found in an extended technical report [10].

With a completed LLVM intermediate representation of the code, we can now continue automated analysis. However, there is already a benefit for a human reverse engineer

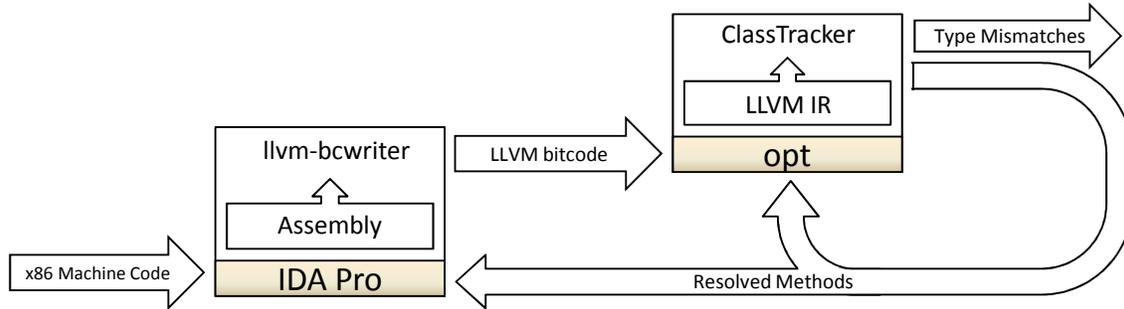


Figure 4: High-Level Architecture of RECALL: Representation of how data flows from x86 assembly through to the LLVM IR for analysis

at this point in the framework. The SSA-based intermediate representation used by LLVM is very easy to read and in many cases will represent vulnerable code more clearly than the x86 assembly equivalent. We thus optionally run `llvm-dis` to generate the textual equivalent of the intermediate representation in a form loadable by IDA Pro, providing an alternate representation of the original binary to assist manual analysis.

5.3. Analyzing the Intermediate Representation

Our RECALL framework includes a plugin for `opt` called `ClassTracker` that performs all of the analysis described in Section 4. It first collects all of the instantiations of new objects by using the heuristics described in Sections 4.1 and 4.2, and it indexes the objects by the relative virtual address of the constructor applied to the new object. Next, it applies the fixed-point reaching definitions algorithm described in Section 4.3, recording all of the possible definitions for each use point (indirect function call site). Third, as calls are made referencing function pointers in a vtable, those function pointers are resolved to their static address. At this point, the congruence check is implicit in that if a call is made to a function pointer that does not exist in the bounds of the vtable, we can report the error. Low-level implementation details of `ClassTracker` are available in a technical report [10].

6. Results

By employing the data flow analysis techniques documented in this paper, we demonstrate that we can increase the effectiveness of existing static analysis techniques on compiled C++ code, as well as identify a class of vulnerability that is often overlooked by existing techniques. Most importantly, all of this analysis can be performed on a compiled binary with no access to the original source. This allows for third parties like software developers, security analysts, or a software consumer software looking to validate

```

"401034":                ; preds = %"401032", %"401029"
...
%25 = getelementptr %0* %24, i32 0, i32 0 ; <i32 (...)*>>
...
call void @29()
...
%31 = getelementptr %0* %30, i32 0, i32 0 ; <i32 (...)*>>
...
call void @35()
...
%37 = getelementptr inbounds %0* %36, i32 0, i32 0 ; <i32 (...)*>>
...
call void @41()
...
%43 = getelementptr inbounds %0* %42, i32 0, i32 0 ; <i32 (...)*>>
...
call void @47()
...
%49 = getelementptr inbounds %0* %48, i32 0, i32 0 ; <i32 (...)*>>
...
call void @53()
...
br label %return

```

Figure 5: Excerpt from LLVM bitcode file generated by `llvm-bcwriter`

code quality to gain a much better view into the code constructs embedded in a compiled binary.

To test our decompilation framework, we created test programs each representing one of the four combinations of stack or heap object declaration and inline or explicit constructor. These programs were compiled without symbols and were provided to IDA Pro as input. In each case, the system was tested for its ability to resolve virtual function calls and to identify instances of type confusion. An example of this process is detailed in Section 6.1. We created a more complex test where an object is declared in one function and referenced in another function, testing the ability of RECALL to perform the analyses described in Section 4 on an interprocedural basis. This test is detailed in Section 6.2.

6.1. Heap-Declared Object, Explicit Constructor

Figure 3a shows an unsafe use of static cast through a void pointer resulting in a vtable escape error. The source

```

class class1 {
public:
    class1();
    ~class1();
    virtual void addRef();
    virtual void print();
    virtual void voidFunc1() { };
    virtual void debug();
};

class class2 {
public:
    class2();
    ~class2();
    virtual void addRef();
    virtual void print();
};

int internalFunction(void *pv) {

    static_cast<class1*>(pv)->addRef();
    static_cast<class1*>(pv)->print();
    static_cast<class1*>(pv)->debug();

    return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
    class1 *C1 = new class1;
    class2 *C2 = new class2;

    internalFunction((void *)C1);
    internalFunction((void *)C2);

    return 0;
}

```

(a) Original Code

```

.text:00401000 sub_401000    proc near
.text:00401000
.text:00401000 mov     eax, [esi]
.text:00401002 mov     edx, [eax]
.text:00401004 mov     ecx, esi
.text:00401006 call   edx
.text:00401008 mov     eax, [esi]
.text:0040100A mov     edx, [eax+4]
.text:0040100D mov     ecx, esi
.text:0040100F call   edx
.text:00401011 mov     eax, [esi]
.text:00401013 mov     edx, [eax+0Ch]
.text:00401016 mov     ecx, esi
.text:00401018 call   edx
.text:0040101A xor     eax, eax
.text:0040101C retn
.text:0040101C sub_401000    endp
.text:0040101C
.text:00401020 _wmain      proc near
.text:00401020 push   esi
.text:00401021 push   edi
.text:00401022 push   8
.text:00401024 call   ??2@YAPAXI@Z
.text:00401029 add     esp, 4
.text:0040102C test   eax, eax
.text:0040102E jz     short loc_401039
.text:00401030 call   sub_401080
.text:00401035 mov     esi, eax
.text:00401037 jmp    short loc_40103B
.text:00401039
.text:00401039 loc_401039:
.text:00401039 xor     esi, esi
.text:0040103B
.text:0040103B loc_40103B:
.text:0040103B push   8
.text:0040103D call   ??2@YAPAXI@Z
.text:00401042 add     esp, 4
.text:00401045 test   eax, eax
.text:00401047 jz     short loc_401061
.text:00401049 call   sub_4010C0
.text:0040104E mov     edi, eax
.text:00401050 call   sub_401000
.text:00401055 mov     esi, edi
.text:00401057 call   sub_401000
.text:0040105C pop     edi
.text:0040105D xor     eax, eax
.text:0040105F pop     esi
.text:00401060 retn
.text:00401061
.text:00401061 loc_401061:
.text:00401061 xor     edi, edi

```

(b) Compiled Binary

Figure 6: Code requiring interprocedural analysis

code shows two heap-declared objects C1 and C2. In the disassembly of the code shown in Figure 3b, we see that the constructors for the two objects are explicitly called at lines 0x010 and 0x029 respectively. Additionally, we can see several calls to virtual functions. Lines 0x03A and 0x043 correspond to the calls to `addRef` and `print` for C1. Lines 0x04B and 0x054 correspond to the calls to `addRef` and `print` for C2. We can see the erroneous call to `debug` at line 0x05D.

After translating the code in Figure 3b to the LLVM IR, it appears as shown in Figure 5. This is an excerpt of the full code that is emitted from `llvm-bcwriter`. The entire output can be found in our technical report [10]. In the LLVM IR, we can see the virtual function calls just as in the disassembly, but it is still just as unclear which function will actually get called. In the LLVM IR, we see the instructions `call void %29` and `call void %35`, which correspond to the calls to `addRef` and `print` for C1. We also see the instructions `call void %41` and `call void %47`, corresponding to the same functions in C2. The final call, `call void %53`, corresponds to the erroneous call

to `debug`.

When the Object Reaching Definition analysis in `ClassTracker` is applied to this code, the various function calls were resolved except for the call to `debug`. In this case there was no corresponding method in the vtable, and `ClassTracker` returns a vtable escape error. The detailed output from `ClassTracker` can be seen in Appendix A.

6.2. Interprocedural Analysis

The previous example shows how `RECALL` is able to translate binary code into the LLVM IR and perform Object Reaching Definition Analysis. However, the vulnerable code that it was able to identify is very unlikely to ever appear in production code. We would hope that this sort of type-casting would be identified through simple code review. Consider the more complex example in Figure 6a: an object is declared in one function and referenced by another function. This code is much more realistic, especially when we consider the possibility that the function `internalFunction` could be called from many differ-

```

"401000":
    . . .
    %2 = getelementptr inbounds %0* %1, i32 0, i32 0 ; <i32 (...)*> [#us
    . . .
    call void %6()
    . . .
    %9 = getelementptr inbounds %0* %8, i32 0, i32 0 ; <i32 (...)*> [#us
    . . .
    call void %13()
    . . .
    %16 = getelementptr inbounds %0* %15, i32 0, i32 0 ; <i32 (...)*> [#
    . . .
    call void %20()
    . . .
    br label %return

```

(a) Excerpt of LLVM bitcode generated by llvm-bcwriter

```

int __cdecl wmain()
{
    int v0; // edi@2
    int v1; // esi@5

    if ( operator new(8u) )
        v0 = sub_401070();
    else
        v0 = 0;
    if ( operator new(8u) )
        v1 = sub_4010A0();
    else
        v1 = 0;
    (**(void (__thiscall **)(__DWORD))v0)(v0);
    (*(void (__thiscall **)(int))(*__DWORD *)v0 + 4)(v0);
    (**(void (__thiscall **)(__DWORD))v1)(v1);
    (*(void (__thiscall **)(int))(*__DWORD *)v1 + 4)(v1);
    (*(void (__thiscall **)(int))(*__DWORD *)v1 + 12)(v1);
    return 0;
}

```

(b) HexRays output

Figure 7: Results after analyzing code from Figure 1b

ent places in the code, each passing in different object types.

In this example, the source code shown in Figure 6a is compiled to the binary representation shown in Figure 6b. Sub_401000 corresponds to the function internalFunction. We can see in this subroutine that it makes three virtual function calls, corresponding to addRef, print, and debug. This binary is provided as input into llvm-bcwriter, which produces the LLVM IR shown excerpted in Figure 7a. The full output can be found in our technical report [10]. Here we can see the the calls to the virtual functions appearing as call void %6, call void %13, and call void %20 respectively.

When running ClassTracker on the IR shown in Figure 6a, the analysis made two passes over the function internalFunction because it is called twice in this code. In the first pass, the virtual function calls were resolved as they belong to class C1. In the second pass, we see a second function pointer assigned to each call site. These are the methods belonging to class C2. Here again ClassTracker reported for the erroneous call to debug. The full output from ClassTracker can be found in Appendix B.

7. Discussion

The capabilities that have been demonstrated in the previous section are advantageous for anyone attempting to reverse engineer C++ compiled binaries. We have already discussed the ability to resolve virtual function calls and identify type confusion issues. In this section, we provide more detail into the benefits of these analyses.

7.1. Static Call Graph Reconstruction

Several existing static analysis techniques require a complete static call graph to operate. The static call graph is a

directed graph comprised of each function as a node and a call location to a function as an edge. As we have seen in this paper, virtual function calls are performed through indirection and make it difficult to identify the corresponding edges of the static call graph. This results in nodes that are completely unconnected and call sites that have incomplete fanout.

By resolving the virtual function calls, we are able to make the static call graph more complete, allowing for previously existing analyses to be increasingly effective. An example of this is dead code elimination, a very common compiler analysis to try to remove unreachable code. Virtual function calls tend to reduce the effectiveness of dead code elimination analyses in that there will be functions strewn around the code that are only reachable through indirection. Since most existing analyses cannot resolve the indirection, for code safety, they must leave any code intact that appears as if it *could* be called.

7.2. Comparison with HexRays

HexRays is a popular, commercially available decompilation plugin for IDA Pro. It reverses binary code into a C-like psuedo-code to make manual analysis easier. Like many decompilation engines, HexRays struggles with C++ constructs. For example, Figure 7b is the HexRays output when decompiling the code from Figure 1b. Here we can see that it is unable to resolve the virtual function calls. Rather, it identifies the first call to the first element in the vtable, but all subsequent calls are left as calls to a base pointer plus an offset.

In comparison, ClassTracker is able to fully resolve all of the virtual function calls. In RECALL, these resolved function calls are populated back into the LLVM IR and the disassembly in IDA as comments next to the call site. The re-

sults from running ClassTracker against the code from Figure 1b are shown in Appendix B.

7.3. Further Vulnerability Identification

This paper presented a data-flow analysis technique and a decompilation framework that allows for the verification of object congruence. This analysis is based on the fundamental concept of reaching definition analysis. There are other basic static analysis techniques that could be developed in this framework to allow for further vulnerability identification. For example, liveness analysis, when applied to C++ objects, would allow us to detect use-after-free conditions.

7.4. False Positives/Negatives

As with any static code analysis, the analyses presented in this paper may incur false positives or negatives. We have identified several potential situations producing false positives:

- We may misclassify a data region as a class when a developer declares a structure on the stack that has the same form as the structure used for C++ objects. For example, if a C struct is declared on the stack that has the same form as that detailed in Figure 2b, then the first heuristic described in Section 4.1 will incorrectly identify this as a C++ object. Fortunately, this does not introduce problems into the further analysis. The vulnerabilities we intend to identify can be introduced through C structs as well, so aggressively assuming a data structure is an object will help identify these cases.
- We may fail to correctly compute the bounds of a vtable. Our system relies on IDA Pro to identify the beginning and end of each vtable, and so our vtable correctness is exactly as good as that of IDA Pro.
- Absent any defects in our implementation, our algorithms should not propagate a class type to a vtable dispatch point unless that propagation is possible.

False negatives are possible as well. As noted in Section 3.3, the identification of new objects is specific to the compile-time development environment. Should a developer link a C++ runtime that does not implement the hints that we use in Section 4.1, we would not identify instantiations of new objects. While the data flow algorithm we present is generic, the heuristics used to create the GEN and KILL sets are specific to a given C++ runtime.

7.5. Analysis of Production Binaries

One of our longer-term goals is to apply our code analysis algorithms to very large applications such as Microsoft Excel and Adobe Reader. Improving our existing implementation so that it can process large software requires additional engineering. We reverse each x86 opcode and operand into an LLVM bitcode format via a manually-written transformation function. Full reversal of a binary file requires implementations of transformation functions for all instructions appearing in the file, and as software becomes large, a greater variety of x86 instructions with complex behaviors appears.

As a result, we opted to test using smaller microbenchmarks that duplicated the vulnerable code constructs present in Excel and Reader while eliding all other code regions with large instruction diversity. Our microbenchmarks elevate the vulnerable code closer to the entry point of the binary. The simulation of the Excel vulnerability contains 2560 bytes of code in the `.text` section, and the Adobe simulation contains 6656 bytes of code in the `.text` section. As such, the subset of the x86 instruction set used becomes far more tractable. Ten more code samples of similar size were tested to ensure each of the different types of object instantiation were able to be identified, and the analysis in each case could be applied interprocedurally.

While further work is required to have full support for production binaries, it is important to note that the specific analyses described in this paper are not impacted by the currently unsupported instructions.

8. Conclusions

C++ is one of the most popular object-oriented development languages in use today. The advancements it provides over the C language has led to its use in base operating system code, applications, device drivers, and more. Its support for templates allowed for the development of the standard template library, which provides a myriad of container types and algorithms that are commonly needed in software development.

However, with all these benefits, come a few drawbacks as well. As we have shown in this paper, standard C++ constructs can lead to the introduction of vulnerable code. Additionally, the manner in which some of these capabilities are achieved create complexities in the low-level machine code that stand in the way of existing compiler analyses.

In this paper, we demonstrated some of the memory safety drawbacks that are introduced by the C++ programming language, and we presented a data-flow analysis that can help remediate these issues. With the decompilation

framework we created as part of this research, implementing these data-flow algorithms is straightforward and effective. With these analysis capabilities, we can ease the understanding of compiled C++ code and identify classes of vulnerable code that have previously gone undetected.

Acknowledgements

We thank David Molnar and the anonymous reviewers for their helpful feedback. This research was supported in part by National Science Foundation contract number CNS-0845309. Any opinions, findings, and conclusions or recommendations expressed in this manuscript are those of the authors and do not reflect the views of the NSF or the U.S. Government.

References

- [1] Adobe. Security update available for Adobe Flash Player. <http://www.adobe.com/support/security/bulletins/apsb11-07.html>, April 2011.
- [2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Not.*, 29:290–301, June 1994.
- [3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1996.
- [4] Bugscam. Bugscam IDC Package. <http://bugscam.sourceforge.net/>.
- [5] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30:775–802, June 2000.
- [6] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Portland, Oregon, 1994.
- [7] Clang. <http://clang.llvm.org/>.
- [8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Los Angeles, California, 1977.
- [9] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.
- [10] D. Dewey and J. Giffin. Static detection of C++ vtable escape vulnerabilities in binary code (extended length). Technical report, School of Computer Science, Georgia Institute of Technology, 2011.
- [11] D. Dhurjati, S. Kowshik, and V. Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.
- [12] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, 2009.
- [13] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest*, 2009.
- [14] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996.
- [15] D. Evans and D. Larochele. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan/Feb 2002.
- [16] S. Heelan. Vulnerability detection systems: Think cyborg, not robot. *IEEE Security & Privacy*, 9(3):74–77, May-June 2011.
- [17] Hey-Rays. <http://www.hex-rays.com/>.
- [18] S. Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, 1977.
- [19] D. Larochele and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
- [20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, California, 2004.
- [21] C. Meadows. A procedure for verifying security against type confusion attacks. In *IEEE Computer Security Foundations Workshop (CSFW)*, Pacific Grove, California, June 2003.
- [22] Microsoft. static_cast Operator. Microsoft Developer Network. <http://msdn.microsoft.com/en-us/library/c36yw7x9%28v%3Dvs.80%29.aspx>.
- [23] Microsoft. Microsoft Security Bulletin MS09-035. Microsoft TechNet, July 2009. <http://www.microsoft.com/technet/security/bulletin/Ms09-035.mspx>.
- [24] Microsoft. Microsoft Security Bulletin MS10-017. <http://www.microsoft.com/technet/security/bulletin/MS10-017.mspx>, March 2010.
- [25] H. Pande and B. Ryder. Data-flow-based virtual function resolution. In *Proceedings of the Third International Symposium on Static Analysis (SAS)*, 1996.
- [26] H. D. Pande and B. G. Ryder. Static type determination for C++. In *Proceedings of the 6th USENIX C++ Technical Conference*, 1994.
- [27] Pete Becker. Working Draft, Standard for Programming Language C++. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>.
- [28] P. Sabanal and M. Yason. Reversing C++. In *Proceedings of BlackHat DC*, 2007.
- [29] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
- [30] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2011.

- [31] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, 2008.
- [32] Sparse. https://sparse.wiki.kernel.org/index.php/Main_Page.
- [33] Splint. <http://splint.org/>.
- [34] B. Stroustrup. *The Design and Evolution of C++*. Pearson Education, 1994.
- [35] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC)*, 2000.
- [36] Wikipedia. [static.cast](http://en.wikipedia.org/wiki/Static_cast). http://en.wikipedia.org/wiki/Static_cast.
- [37] Y. Xie, A. Chou, and D. Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the European Software Engineering Conference (ESEC)*, 2003.

A ClassTracker Output

The following is the output from ClassTracker when run against the LLVM IR shown in Figure 5.

```
Completed enumeration of class: sub_401070
Completed enumeration of class: sub_4010A0
=====
call void @29()
Tracked Back To: %11 = phi %0** [ %9, %"401019" ],
[ %Class0, %"401010" ] ; <%0**> [#uses=2]
WARNING: Multiple class definitions reach use
Tracking Class: Class0
Tracked Back To: %Class0 = alloca %0* ; <%0**> [#uses=2]

call void @29() #sub_4010B0
=====
call void @35()
Tracked Back To: %11 = phi %0** [ %9, %"401019" ],
[ %Class0, %"401010" ] ; <%0**> [#uses=2]
WARNING: Multiple class definitions reach use
Tracking Class: Class0
Tracked Back To: %Class0 = alloca %0* ; <%0**> [#uses=2]

call void @35() #sub_401080
=====
call void @41()
Tracked Back To: %23 = phi %0** [ %21, %"401032" ],
[ %Class1, %"401029" ] ; <%0**> [#uses=3]
WARNING: Multiple class definitions reach use
Tracking Class: Class1
Tracked Back To: %Class1 = alloca %0* ; <%0**> [#uses=2]

call void @41() #sub_4010B0
=====
call void @47()
Tracked Back To: %23 = phi %0** [ %21, %"401032" ],
[ %Class1, %"401029" ] ; <%0**> [#uses=3]
WARNING: Multiple class definitions reach use
Tracking Class: Class1
Tracked Back To: %Class1 = alloca %0* ; <%0**> [#uses=2]

call void @47() #sub_4010C0
=====
call void @53()
Tracked Back To: %23 = phi %0** [ %21, %"401032" ],
[ %Class1, %"401029" ] ; <%0**> [#uses=3]
WARNING: Multiple class definitions reach use
Tracking Class: Class1
Tracked Back To: %Class1 = alloca %0* ; <%0**> [#uses=2]

ERROR: Call to offset outside bounds of vtable
```

B ClassTracker Output

The following is the output from ClassTracker when run against the LLVM IR shown in Figure 7a.

```
Completed enumeration of class: sub_401080
Completed enumeration of class: sub_4010C0
=====
call void @6()
Tracked Back To: %Arg_esi_addr = alloca i32 ; <i32*> [#uses=6]
Tracked Back To: %19 = call i32 @sub_401000(i32 %18) ; <i32> [#uses=0]
Tracked Back To: %11 = phi %0** [ %9, %"401039" ],
[ %Class0, %"401030" ] ; <%0**> [#uses=2]
WARNING: Multiple class definitions reach use
Tracking Class: Class0
Tracked Back To: %Class0 = alloca %0* ; <%0**> [#uses=2]

call void @6() #sub_401090
=====
call void @13()
Tracked Back To: %Arg_esi_addr = alloca i32 ; <i32*> [#uses=6]
Tracked Back To: %19 = call i32 @sub_401000(i32 %18) ; <i32> [#uses=0]
Tracked Back To: %11 = phi %0** [ %9, %"401039" ],
[ %Class0, %"401030" ] ; <%0**> [#uses=2]
WARNING: Multiple class definitions reach use
Tracking Class: Class0
Tracked Back To: %Class0 = alloca %0* ; <%0**> [#uses=2]

call void @13() #sub_4010A0
=====
call void @20()
Tracked Back To: %Arg_esi_addr = alloca i32 ; <i32*> [#uses=6]
Tracked Back To: %19 = call i32 @sub_401000(i32 %18) ; <i32> [#uses=0]
Tracked Back To: %11 = phi %0** [ %9, %"401039" ],
[ %Class0, %"401030" ] ; <%0**> [#uses=2]
WARNING: Multiple class definitions reach use
Tracking Class: Class0
Tracked Back To: %Class0 = alloca %0* ; <%0**> [#uses=2]

call void @20() #sub_4010B0
=====
call void @6()
Tracked Back To: %Arg_esi_addr = alloca i32 ; <i32*> [#uses=6]
Tracked Back To: %21 = call i32 @sub_401000(i32 %20) ; <i32> [#uses=0]
Tracked Back To: %Class1 = alloca %0* ; <%0**> [#uses=2]

call void @6() #sub_401090
#sub_401090
=====
call void @13()
Tracked Back To: %Arg_esi_addr = alloca i32 ; <i32*> [#uses=6]
Tracked Back To: %21 = call i32 @sub_401000(i32 %20) ; <i32> [#uses=0]
Tracked Back To: %Class1 = alloca %0* ; <%0**> [#uses=2]

call void @13() #sub_4010A0
#sub_4010D0
=====
call void @20()
Tracked Back To: %Arg_esi_addr = alloca i32 ; <i32*> [#uses=6]
Tracked Back To: %21 = call i32 @sub_401000(i32 %20) ; <i32> [#uses=0]
Tracked Back To: %Class1 = alloca %0* ; <%0**> [#uses=2]

ERROR: Call to offset outside bounds of vtable
```