# Uncovering Use-After-Free Conditions In Compiled Code

David Dewey[1] ddewey@gatech.edu
Bradley Reaves[2] reaves@ufl.edu
Patrick Traynor[2] traynor@cise.ufl.edu

[1]School of Computer Science, Georgia Institute of Technology
[2]Department of Computer and Information Science and Engineering, University of Florida

***Abstract*—Use-after-free conditions occur when an execution path of a process accesses an incorrectly deallocated object. Such access is problematic because it may potentially allow for the execution of arbitrary code by an adversary. However, while increasingly common, such flaws are rarely detected by compilers in even the most obvious instances. In this paper, we design and implement a static analysis method for the detection of use-after-free conditions in binary code. Our new analysis is similar to available expression analysis and traverses all code paths to ensure that every object is defined before each use. Failure to achieve this property indicates that an object is improperly freed and potentially vulnerable to compromise. After discussing the details of our algorithm, we implement a tool and run it against a set of enterprise-grade, publicly available binaries. We show that our tool can not only catch textbook and recently released in-situ examples of this flaw, but that it has also identified 127 additional use-after-free conditions in a search of 652 compiled binaries in the Windows system32 directory. In so doing, we demonstrate not only the power of this approach in combating this increasingly common vulnerability, but also the ability to identify such problems in software for which the source code is not necessarily publicly available.**

## I. Introduction

Many software developers have the unenviable task of maintaining large, often complex codebases. As the number of lines of code increases, it becomes virtually impossible for any single developer to be able to reason about their software's behavior, yet alone its security. As such, large software projects commonly have a range of subtle vulnerabilities.

Use-after-free vulnerabilities represent one particularly difficult example of this problem. The result of a programmer freeing a memory resident object that may be needed on some other code execution path creates an opportunity for an adversary to inject and execute arbitrary code. Not only do current compilers generally fail to catch even the most trivial instances of this problem, but developers relying on opaque binary libraries may be further endangering the security of their software. Such vulnerabilities are now regularly being exploited in the wild [19]–[21].

In this paper, we develop a static analysis technique for detecting use-after-free conditions. We make a clear distinction between use-after-free conditions and vulnerabilities because static analysis techniques cannot always determine the real-world exploitability of a given code flaw. Just as compiler warnings imperfectly alert developers to common errors, indications of use-after-free conditions allow an analyst to focus on the most likely exploitable code.

Our approach is based on a technique from compiler theory known as available expression analysis. While this technique has traditionally been used to make code more efficient via common subexpression elimination, we extend this concept to instantiated objects to show that an object that does not exhibit this property has been improperly freed and is potentially vulnerable. After formally describing our available object definition analysis, we describe our implementation of the algorithm, which applies our techniques to binary code. In so doing, we make the following contributions:

- **Develop a general-purpose data flow algorithm for detection of use-after-free conditions:** We present and formally define a data flow algorithm based on a technique from compiler theory known as available expression analysis. Our technique, called Available Object Definition Analysis (AODA) can be used by source code analysis tools, compilers and binary static analysis frameworks to identify use-after-free conditions.
- **Programmatically identify use-after-free conditions in compiled binaries:** We implement our algorithm for use on compiled binaries, as source code for the majority of commercial software is not publicly available. We also discuss the research challenges of the implementation.
- **Confirm known vulnerabilities and discover many potential new ones:** We use our tool to confirm both exemplar and known instances of use-after-free vulnerabilities. We then analyze 652 popular binaries in the Windows system32 directory and identify 127 new use-after-free conditions in these files.

The analyses presented in this paper focus specifically on use-after-free conditions in compiled C++ code. While this same vulnerability can be present in other languages, they tend not to present the opportunity for such formulaic exploitation as is described in Section III-B. With that, they have historically been less exploitable, and do not pose nearly the level of threat that is seen with C++.

We focus on compiled code because our research agenda is focused on identifying vulnerabilities in commercial software, including applications like Microsoft Excel or Google Chrome, libraries like mshtml.dll (Microsoft's HTML parser) and acrord32.dll (Adobe's PDF parser), and operating systems including Microsoft Windows, without access to the original source code. The importance of auditing compiled binaries cannot be overstated. Binary analysis is how users and consumers of libraries, applications, and operating systems verify and contribute to the security of the software they trust. Security consultants and researchers regularly use disassembly tools to reverse engineer commercial software to identify security vulnerabilities. The famous (yet now defunct) Full Disclosure mailing list was populated by vulnerabilities identified from binary analysis, and a significant portion of CVEs are attributed to security researchers working only with compiled binaries. Accordingly, binary analysis has been and will continue to be a significant driver of software security.

## II. Related Work

The programmatic identification and elimination of code flaws has been studied since the beginnings of software engineering. Much of this effort has centered around the static analysis of source code. Tools such as Lint [16] (and its modern day implementation, Splint [29]) and Sparse [28] are popular for finding flaws in kernel and userland code in Linux. Similarly, Clang [5] is included with Apple's Xcode. These tools implement detection techniques for a number of weaknesses, including buffer overflows [12], [17] and format string vulnerabilities [25]. Constraint solving tools such as ARCHER [32] determine the safety of behaviors such as array access. Still other tools can help identify pointer access errors [1]. Searching specifically for use-after-free vulnerabilities, Caballero, et al. developed *Undangle* [3]; a runtime taint tracking tool used to detect dangling pointers. Also focused on dangling pointers, Lee, et al. created DangNull to nullify class pointers when objects are deleted. While this is a sound solution to prevent exploitation of use-after-free vulnerabilities, it relies on the appliication's pre-existing ability to handle null pointers, else it would introduce stability problems. Tice [30] proposed a compile time solution to verify the validity of a virtual function pointer before its invocation via inserting verification checks.

Many analysis tools have been integrated directly into popular compilers to provide developers with warnings and errors at compile time. C++ analyses typically come in the form of checks for type, "const"-ness and volatility, all of which are fully enumerated in the C++ standard [24]. Significant research has attempted to extend required checks with virtual function call resolution in C++ programs. Bacon and Sweeny [2], 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, the call 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 [22], [23] and Calder and Grunwald [4] expand this concept to eliminate late binding where possible to take advantage of instruction pipelining on modern-day processors. SAFECode, a system developed by Dhurjati et al. [9], introduces a new type system that can be enforced at compile-time to prevent a range of vulnerabilities. However, in spite of great progress in this area, many problems remain unsolved [13].

All of the aforementioned tools require access to the source code of the potentially vulnerable program. One of the major motivations for the work presented in this paper is to be able to analyze code that has already been distributed to the public. In these cases, the analysis must function on compiled binaries.

Because there are many scenarios where a security analyst must audit software without source code, binary decompilation has been studied extensively. Such analysis often requires the transformation of binary code to an intermediate representation. Cousot and Cousot showed that restructuring a language into such an abstract representation allows for the simplified implementation of many complex analyses [6]. This observation has been extended and implemented by a number of open and closed source tools. Song et al. developed one of the first practical frameworks for binary decompilation and analysis with Bitblaze [27]. The commercially popular Hex-Rays plugin for IDA Pro reverses binary code to a C-like intermediate representation [14], and Dullien and Porst developed the Reverse Engineering Intermediate Language (REIL) for their commercial product, Bindiff [11]. Yakdan et al. [?] offer a novel method for decompilation that eliminates the excessive use of goto's that are often introduced. This paper builds upon the Recall decompilation framework by Dewey and Giffin [8], which reverses binary code to the popular LLVM intermediary representation [18].

One of the core requirements for our use-after-free analysis is the proper identification of C++ objects as they are represented in binary code. Such binary data structure recovery has been covered extensively in the literature. For example, Dolan-Gavitt et al. [10] 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 [7], a system that uses Bayesian unsupervised learning to detect the presence of data structures in memory indicative of a bot infection. In the area of reverse engineering tools, Slowinska et al. [26] created a system that recovers data structures from a compiled binary. This work relies on the heuristics defined by Dewey et al. [8], which are discussed in greater detail in Section V.

## III. Background

In this paper, we present a static analysis technique to detect use-after-free conditions in compiled binaries. Because the structure of how C++ objects are compiled and stored in memory is crucial to both the understanding of use-after-free vulnerabilities as well as analysis of C++ binaries, we give a brief description of how C++ objects are represented after compilation. We then give a short description of how use-after-free vulnerabilities occur in code and how they are exploited. We conclude this background section with a review of available expression analysis from compiler theory.

### A. *C++ Objects in Memory*

One of the tasks of a C++ compiler is to ensure the correct storage of C++ classes in memory while providing both multiple inheritance and virtual functions. C++ compilers store instantiated objects as contiguous structures in memory.

This is fortunate because it simplifies the discovery of object instantiations in compiled binaries. Our discussion here is focused on how Visual Studio compiles C++; other compilers represent C++ objects in a similar manner.

Classes without virtual functions are represented in memory like traditional C structs. Simply, the properties (data members) of the object are placed in contiguous memory in order of declaration in the code. The object's method calls are compiled as direct calls to functions that take a pointer to the object as an argument (the `this` pointer).

When an object has one or more virtual functions, virtual function calls must be dispatched at runtime through the use of a virtual function table (vtable). A vtable for an object is a contiguous set of function pointers that point to the appropriate functions for that object type. The virtual function can be called by finding the correct function pointer in the function pointer table.

Objects with virtual methods are laid out in memory the same way as objects without virtual methods, but have an initial member element: a pointer to the class's vtable. This is a pointer to an array of function pointers (one for each virtual funciton) stored in the read-only .data section of the binary. When a class inherits from a single base class, the class instance begins with a vtable pointer, followed by the base object properties and then the derived object properties. When a class inherits from more than one class, the base class layouts are placed consecutively, with the derived class's data members following. Classes that are the result of multiple inheritance have more than one vtable — one for each base class. Virtual methods from the derived class will be appended to the vtable for the first base class.

The memory structures described above are created on the stack or the heap by the class constructor depending on how the object is allocated in the source code. The compiler may also choose to make the constructors inline or create a separate function call for the constructor. In Visual Studio, this is a configurable optimization that can be set by the developer.

```
00:     class A {
01:     private:
02:             int reference;
03:
04:     public:
05:             A();
06:             ~A() { };
07:             virtual void addRef();
08:             virtual void print();
09:     };
10:
11:     int main(int argc, char* argv[])
12:     {
13:             A *a = new A;
14:
15:             a->addRef();
16:             a->print();
17:
18:             if (atoi(argv[1]) == 1) {
19:                     delete a;
20:             }
21:
22:             //. . .
23:             //Memory allocation operations
24:             //. . .
25:
26:             a->print();
27:
28:             return 0;
29:     }
```

Fig. 1: Sample C++ code with a use-after-free vulnerability. Note that the presence of a command line argument at line 18 causes object `a` to be deleted, even though it is called again at line 26.

## B. Use-After-Free Vulnerabilities

Use-after-free (UAF) vulnerabilities are a class of software flaws that involve using a memory resident object after it has been freed. UAF vulnerabilities most commonly occur when a C++ object that was allocated on the heap is accessed after it is deleted, but stack-allocated objects can also be used after a free.

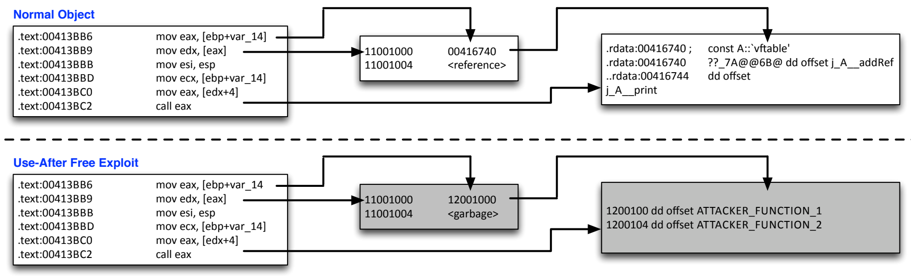
Developers can easily make this memory management error, especially in large and complex codebases, and often in an attempt to prevent memory leaks. This creates a condition where an object is deleted on some code paths, but not all. The code in Figure 1 provides a simplified example. In this example, the developer chose to delete the object if the first command line argument is "1", but needed the object under all other conditions. If an attacker can cause the program to take the first branch, the object is deleted, but then is used later in the program.

When an object is deleted, new data takes the place of the previously deleted object's properties and/or vtable pointer. Later, if the deleted object's property is accessed, this new data could be read or modified, affecting the reliable operation of the program. If a deleted object's virtual method is called, whatever data has been written to the old vtable pointer will be dereferenced to locate the correct function pointer. This can result in memory access violations or otherwise unstable execution.

In addition to the hazards of a normally functioning program with a use-after-free condition, it can also be a path to software exploitation. While a use-after-free can result in an attacker with the ability to write data to the stack or heap to influence decisions made based on an object's properties, the greatest danger is if the use-after-free includes a virtual function call. In that case, the attacker can write his own vtable containing pointers to shellcode and fill the deleted object's memory with references to that vtable. On the virtual function call, execution will transfer to the attacker's injected code.

Figure 2 demonstrates a use-after-free exploit of the code shown in Figure 1. The left side of Figure 2 shows a fragment of the compiled code from Figure 1. That fragment completes with a call to the virtual function `print` at line 413BC2. The object pointer is consulted to find the pointer to the vtable at line 413BB9. At runtime, the object is stored on the heap — in this example, the object is stored at address 11001000, and the class's vtable is located at address 416740. The function pointer stored at offset 4 represents the second virtual function declared in the class (since function pointers are 4 bytes each).

If this object were overwritten using a use-after-free vulnerability, and the attacker could control the value of the data at the heap location where the object was stored (which in many cases is not difficult), the attacker could overwrite the vtable pointer with a pointer to the attacker's own vtable. The bottom half of Figure 2 shows this scenario with gray boxes. In that example, the attacker has established a vtable pointer of 12001000 (also heap data written by the attacker) and has preloaded that location with function pointers to malicious



```
.text:00413BB6        mov eax, [ebp+var_14]
.text:00413BB9        mov edx, [eax]
.text:00413BBB        mov esi, esp
.text:00413BBD        mov ecx, [ebp+var_14]
.text:00413BC0        mov eax, [edx+4]
.text:00413BC2        call eax
```

```
11001000        00416740
11001004        <reference>
```

```
.rdata:00416740 ;    const A::`vftable'
.rdata:00416740      ??_7A@@6B@ dd offset j_A__addRef
..rdata:00416744     dd offset
j_A__print
```

```
.text:00413BB6        mov eax, [ebp+var_14
.text:00413BB9        mov edx, [eax]
.text:00413BBB        mov esi, esp
.text:00413BBD        mov ecx, [ebp+var_14]
.text:00413BC0        mov eax, [edx+4]
.text:00413BC2        call eax
```

```
11001000        12001000
11001004        <garbage>
```

```
1200100 dd offset ATTACKER_FUNCTION_1
1200104 dd offset ATTACKER_FUNCTION_2
```

Fig. 2: Memory layout of a valid object and after a use-after-free exploit

functions (again, written by the attacker). When the second function in the vtable is called, the processor will call address `dd offset ATTACKER_FUNCTION2` instead of address `dd offset j_A__print`.

Unfortunately, compilers fail to detect use-after-free vulnerabilities. Not only do they miss complex flaws that are the result of interprocedural memory management, they also miss trivial examples like the one in Figure 1. Visual Studio 2012, g++ 4.8.3, and clang++ 3.5 *all fail to detect the flaw in that exact code sample* (outputs shown in [**?**]).

## C. Available Expression Analysis

Available expression analysis (abbreviated AVAIL) is a common compiler data flow analysis that enables common subexpression elimination. AVAIL takes as input a control flow graph consisting of basic blocks, and AVAIL identifies, for each basic block, a set of expressions that will always be computed before the entry of a given basic block. Knowing the set of expressions that are always computed before a basic block allows a compiler to set aside results to avoid redundant computation (specifically, common subexpressions). While AVAIL is usually presented as a local (within a single procedure) algorithm, interprocedural available expression analysis is also possible.

More specifically, AVAIL computes four sets for each basic block $B$: GEN$[B]$, KILL$[B]$, AVAIL$_{\text{IN}}[B]$, and AVAIL$_{\text{OUT}}[B]$. AVAIL$_{\text{IN}}$ and AVAIL$_{\text{OUT}}$ refer to the expressions that are available before and after each block (respectively). GEN is the set of new expressions that are defined in the basic block, and KILL is the set of expressions whose values have changed because a variable used in the expression has changed. These sets are computed with the following two relations:

$$\text{AVAIL}_{\text{IN}}[B] = \bigcap_{p\ proceeds B} \text{AVAIL}_{\text{OUT}}[p] \qquad (1)$$

$$\text{AVAIL}_{\text{OUT}}[B] = \text{GEN}[B] \cup (\text{AVAIL}_{\text{IN}}[B] - \text{KILL}[B]) \qquad (2)$$

where $p$ is a basic block (that proceeds a block $B$). In our analysis, objects must be tracked interprocedurally. In those cases, AVAIL$_{\text{IN}}$ at the entry of a function $F$ is equal to AVAIL$_{\text{OUT}}$ at the call site to $F$ from a call site $c$. Within a basic block, we say that an expression is in the "AVAILset" at an execution point $a$ if it has been generated before $a$, or if the expression is in AVAIL$_{\text{IN}}$ and not killed before point $a$.

These relations, when written for every basic block, form a system of equations. Because the control flow graph contains back-edges (from loops and other control structures), and thus blocks can be affected by both preceding and subsequent blocks, solving the system of equations requires a fixed-point algorithm (that runs through each block iteratively until the four sets for every block do not change).

The recurrence relationship described above forms the basis of many other data flow analysis algorithms, with slightly different definitions of IN, OUT, GEN, KILL, and the so-called "meet" operator, which in the case of AVAIL is set union. In this work, we define a new data-flow analysis algorithm in the following section. Our algorithm uses the same recurrences above, but with new definitions of GEN and KILL that allow us to track object instantiation and deletion rather than availability of expressions.

## IV. Available Object Definition Analysis

As discussed in the previous section, use-after-free vulnerabilities occur when an object is accessed after being freed. Frequently, the object is freed along some code paths, but not others. Accordingly, use-after-free errors can be detected by finding every access where the object may have been deleted on some proceeding code path. Thus, an analysis that determines if all code paths to an access contain a valid object definition will detect use-after-free conditions. In this section, we define such an analysis and term it Available Object Definition Analysis (AODA).

Our crucial insight is that the recurrence relations from AVAIL can be used to track the availability of instantiated objects to detect use-after-free errors. To track instantiated objects, we simply need to redefine how the GEN and KILL sets are populated during the analysis. We will use the same notation for these sets, but note their alternative meanings in AODA. In AODA, GEN is the set of objects that are *instantiated* in a basic block, while KILL is the set of objects that are *freed* in a basic block. AVAIL$_{\text{IN}}$ is simply the set of objects that are instantiated (and thus, available) along all code paths before a basic block, while AVAIL$_{\text{OUT}}$ are the objects that are available after a basic block has executed.

Tracking all instantiated class pointers is significantly more difficult than tracking expressions. The following section details how we reliably compute the GEN and KILL sets. Given GEN and KILL, computing AVAIL$_{\text{IN}}$ and AVAIL$_{\text{OUT}}$ requires a straightforward application of iteration over the control flow graph. We rely on the RECALL framework to compute the control flow graph, and identify object instantiations and deletions.

Although in this paper we evaluate AODA on compiled binaries, as a data flow algorithm it can also be implemented within a compiler or within a stand-alone static analysis tool.

## V. IMPLEMENTATION

In this section, we describe how we extended the RECALL binary analysis framework to implement AODA. This section includes the technical details involved in identifying object instantiation, usage, and deletion. In the following section, we use this implementation to detect use-after-free conditions in several test applications adapted from pubilcly available projects as well as 652 Microsoft Windows libaries and a previously disclosed use-after-free vulnerability.

### *A. Assumptions*

As discussed in Section I, our research focuses on the common and important problem of identifying software vulnerabilities in compiled code. Because we focus on commercially available, release-build code, we make no claims about the applicability of our algorithms or implementation to malware or other hardened, obfuscated code. We chose to focus on Windows binaries compiled to x86 by Microsoft Visual Studio for this paper. We chose this platform because the majority of closed-source software (which requires binary analysis) is implemented for that platform. While our platform choice influences our implementation and choice of experimental targets, the techniques we present will be applicable to any x86 binary format and compiler choice. For the sake of generality, we assume that Runtime Type Information (RTTI) is unavailable to the analysis. If this information is available, it can only make object detection (and our subsequent analysis) more accurate.

### *B. Identifying object instantiation and deletion*

To compute GEN for AODA, we must identify every object instantiation, and to compute KILL for AODA we must identify every object deletion. When C++ code is compiled, the clear type definitions in the source code become calls to relevant constructors that create the memory objects described in Section III-A, and accordingly are non-trivial to identify. We rely on RECALL's ClassTracker to identify C++ object creation and deletion with high reliability. Identifying object creation is prior work [8], and accordingly we provide only a rough sketch of this analysis. We refer the reader to [8] for the complete details and justifications of correctness.

Briefly, RECALL uses four simple, reliable heuristics to identify C++ objects that reflect *all possible ways* a C++ object can be instantiated at runtime:

- Stack-allocated object with in-line constructor
- Heap-allocated object with in-line constructor
- Stack-allocated object with called constructor
- Heap-allocated object with called constructor

The heuristics for detecting heap objects rely on the compiler's use of a specific function for the `new` operator. In the case of Visual Studio, after compilation this function is referred to as YAPAXI.

Once an object instantiation is located, RECALL discovers an object's type by tracing the size of the new vtable and set of funtion pointers as well as the number and size of the properties of the object. When a new object is encountered, the virtual address of the constructor of the object is used as an opaque identifier for that object type.

We extend RECALL to also detect object deletion. Deletion detection also differs between stack and heap declared objects. Stack-alloocated objects are deleted by first calling the object's destructor, then calling a `delete` operator (named YAXPAX Visual Studio after compilation). The compiler automatically deletes stack-declared objects when they fall out of scope.

Heap-allocated objects are deleted only when the developer makes an explicit call to `delete`. When this occurs, Visual Studio creates a helper function that calls the object's destructor and the delete function YAXPAX. The name of the helper function will include the string `_scalar_deleting_destructor` or `_vector_deleting_destructor` depending on whether the developer calls `delete` or `delete[]`, respectively. In the case of heap-allocated objects, to ensure correctness RECALL traverses into the helper function to ensure that the class destructor is called and that there is an explicit call to YAXPAX. RECALL detects deletion of both types of object. In particular, because heap object deletion is so distinctive, we are highly confident in our ability to detect deleted objects on the heap.

### *C. Complex Real-World Scenarios*

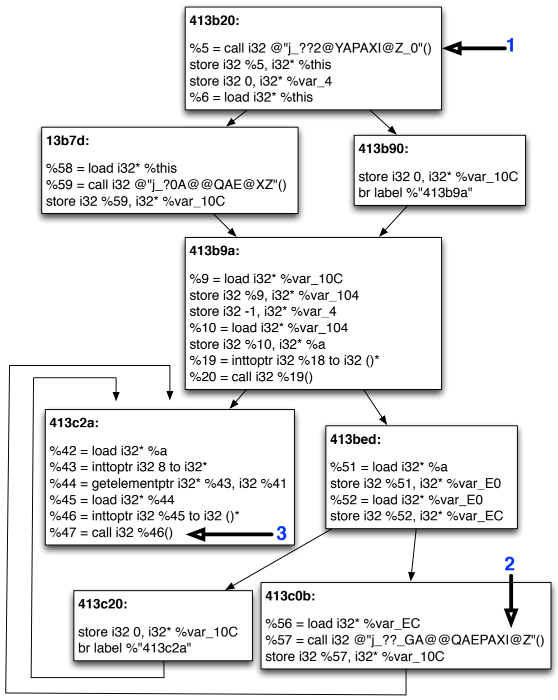
While the basic concept of how AODA is able to identify object instantiations and deletions is covered above, in practice, developers employ many variations on this concept that complicate the analysis. The details of some of the more complex, real-world scenarios we identified are covered in the following subsections.

*1) Virtual Destructors:* A developer may choose to declare the destructor for a class as `virtual`. This is important if the developer encounters a scenario where they wish to delete an instance of a derived class through a pointer to a base class. This however, introduces complexity for AODA. The problem is that when performing the data flow analysis, there is no clear call to the destructor for the object. Rather, an indirect function call is inserted into the binary. The structure of the object must be fully recovered to gain an understanding of what that indirect function call is doing. We use the RECALL framework [8] to reconstruct the vtable of the object. Then, any indirect function call encountered during the data flow analysis is reconciled to its actual function, and further analyzed to determine whether it is actually the destructor for the object.

Of course, following every indirect function call during the analysis can cause the runtime of the analysis to grow exponentially. To combat this issue, AODA takes a configuration parameter to define how deeply it should follow the indirect calls to look for the destructor. In practice, we found that a depth of one is sufficient to find virtual destructors. To further limit the impact of following indirect function calls, this traversal only occurs for functions which employ the `__thiscall` calling convention (i.e., one that passes the `this` pointer as an argument in the `ecx` register). This ensures that the indirect function call is, in fact, a call to a class method and not some other arbitrary function.

*2) Factory Design Pattern:* Much like virtual destructors require AODA to traverse function calls to identify a destructor, the factory design pattern requires AODA to traverse function calls to identify the instantiation of new objects. The factory design pattern is a common object-oriented design construct in which the developer creates a method which will instantiate differing objects based on the arguments specified. When this design pattern is used, the constructor for the object is not directly encountered during the data flow analysis. Rather, AODA must follow function calls to determine whether the returned value is a class pointer.

In the same way AODA takes an argument to specify the depth to which it should traverse indirect calls to find destructors, the depth of function calls to be traversed to find constructors is configurable. Again, in practice, a depth of one was found to be sufficient. However, it is more difficult to limit the functions that will be traversed to find instantiated objects. When the constructor depth is set to one, it will follow every function call encountered during the data flow analysis to determine whether a class pointer is returned.



```
413b20:

%5 = call i32 @"j_??2@YAPAXI@Z_0"()
store i32 %5, i32* %this
store i32 0, i32* %var_4
%6 = load i32* %this
```

```
13b7d:

%58 = load i32* %this
%59 = call i32 @"j_?0A@@QAE@XZ"()
store i32 %59, i32* %var_10C
```

```
413b90:

store i32 0, i32* %var_10C
br label %"413b9a"
```

```
413b9a:

%9 = load i32* %var_10C
store i32 %9, i32* %var_104
store i32 -1, i32* %var_4
%10 = load i32* %var_104
store i32 %10, i32* %a
%19 = inttoptr i32 %18 to i32 ()*
%20 = call i32 %19()
```

```
413c2a:

%42 = load i32* %a
%43 = inttoptr i32 8 to i32*
%44 = getelementptr i32* %43, i32 %41
%45 = load i32* %44
%46 = inttoptr i32 %45 to i32 ()*
%47 = call i32 %46()
```

```
413bed:

%51 = load i32* %a
store i32 %51, i32* %var_E0
%52 = load i32* %var_E0
store i32 %52, i32* %var_EC
```

```
413c20:

store i32 0, i32* %var_10C
br label %"413c2a"
```

```
413c0b:

%56 = load i32* %var_EC
%57 = call i32 @"j_??_GA@@QAEPAXI@Z"()
store i32 %57, i32* %var_10C
```

Fig. 3: Control flow graph of code from Figure 1. At point 1, the object `a` is declared. At point 2, the object is deleted. At point 3, `a->print()` is called again. Note that because point 3 is reachable via point 2, a use-after-free vulnerability is possible.

### *D. Computing AODA and Finding Use-After-Free vulnerabilities*

The second pass made by RECALL over the intermediate representation performs the fixed point availability analysis algorithm described in Section IV. In this pass, it performs a forward analysis iterating over each basic block. As object instantiation points are identified, the virtual address of the constructor is added to the GEN set for the given basic block. As object deletion points are identified, objects are added to the KILL set. The analysis identifies which particular object is deleted by following the use-def chain to locate its instantation. As RECALL iterates to the next basic block, the objects in all the $\text{AVAIL}_{\text{OUT}}$ sets of the block's predecessors are added to the $\text{AVAIL}_{\text{IN}}$ set for the current block. This process is repeated in a fixed-point algorithm until $\text{AVAIL}_{\text{IN}}$ and $\text{AVAIL}_{\text{OUT}}$ sets have been generated for each basic block.

In the third pass over the code, RECALL identifies usage points of binary objects by using a forward analysis that iterates over each basic block in the IR. In each basic block, RECALL looks for indirect function calls — calls to function pointers that are dereferenced from a larger containing data structure (i.e., a vtable). When RECALL identifies this condition, it traverses the use-def chain of the function pointer backwards to identify the instantiation point of the containing structure. If the instantiation point can be traced back to an object instantiation detected in the first pass, RECALL then determines whether the virtual address of the call to that object's constructor is present in the AVAIL set for the basic block at the program point where the usage occurs. If the call to the object's constructor is not present in the AVAIL set for the basic block where the usage occurs, a potential use-after-free condition has been discovered.

## VI. RESULTS

After implementing the Availabile Object Definition Analysis described in Section IV, we tested the framework on a series of simple examples to ensure the algorithm was operating correctly. The process used for testing and the results of the tests are detailed in Section VI-A. Then, use-after-free conditions were injected into several publicly available projects as detailed in Section VI-B. By scanning these projects, we ensure that our analysis is correct for real-world applications. To ensure the analysis would work on a real-world vulnerability, it was run against a library with a known use-after-free vulnerability. Section VI-B details how the results of the analysis were verified. Once we verified the results on a control set of binaries, the framework was run over a substantial subset of the the .dll's in the system32 directory on a default install of Windows 7. Section VI-D details the number of use-after-free conditions that exist unpatched on one of the world's most popular operating systems.

### *A. Control Set Results*

The first step in verifying the Available Object Definition Analysis was to test it on a series of simple examples and manually verify the results. An example of one of the simple

```
Processing Function: _main(i32, i*)
Processing Basic Block: 413b20
    Avail_in = {}
    Avail_out = {j_??0A@@QAE@X}    ## 1 ##
Processing Basic Block: 413b7d
    Avail_in = {j_??0A@@QAE@X}
    Avail_out = {j_??0A@@QAE@X}
Processing Basic Block: 413c2a
    Avail_in = {}                ## 3 ##
    Avail_out = {}
Processing Basic Block: 413bed
    Avail_in = {j_??0A@@QAE@X}
    Avail_out = {j_??0A@@QAE@X}
Processing Basic Block: 413c0b
    Avail_in = {j_??0A@@QAE@X}    ## 2 ##
    Avail_out = {}
Processing Basic Block: return
    Avail_in = {}
    Avail_out = {}
Avail sets complete...

Processing Basic Block: 413b90
Processing Basic Block: 413b9a
    Found use of: j_??0A@@QAE@X
    In Avail Set...
    Found use of: j_??0A@@QAE@X
    In Avail Set...
Processing Basic Block: 413c2a
    Found use of: j_??0A@@QAE@X
    ERROR - Not in Avail Set...
    Potential Use After Free Condition
Processing Basic Block: 413bed
Processing Basic Block: return
```

Fig. 4: AODA results show that compiled code of Figure 1 contains a use-after-free vulnerability

```
Avail sets complete...
Processing Basic Block: ed0c85e
Processing Basic Block: ed0c877
Processing Basic Block: ed0c881
Processing Basic Block: ed0c887
Processing Basic Block: ed0c883
Processing Basic Block: ed0c8aa
Processing Basic Block: ed0c8b6
Processing Basic Block: ed0c8f5
Processing Basic Block: ed0c8ff
Processing Basic Block: ed0c902
Processing Basic Block: ed0c8f9
    Found use of: ??0CEditSession@@QAE@P6GJKPAV0@@Z@Z
    ERROR - Not in Avail Set...
    Potential Use After Free Condition
```

Fig. 5: AODA results detect use of an object that is not in the AVAIL set in a previously-disclosed vulnerability

examples can be found in Figure 1. This example code was compiled and then run through RECALL, which generated an LLVM bitcode file. The control flow graph of the resulting bitcode can be seen in Figure 3 (some code is omitted). Here we can see the object `a` is instantiated in basic block 413b20 (denoted as "1" in Figure 3). We can then see that the object is deleted in basic block 413c0b (denoted as "2" in Figure 3). Then the object is used in basic block 413c2a (denoted as "3" in Figure 3). However, since there is a path to 413c2a whereby the object is deleted, we have a potential use-after-free vulnerability.

When AODA is run on the control flow graph in Figure 3, it automates the detection logic described above. It does this in two passes over the code. The first pass identifies the AVAILsets as shown in Figure 4. The most notable points of the first pass are where the analysis detects the generation of a new object in basic block 413b20 (denoted as "1" in Figure 4), and where it detects an object being killed in basic block 413c0b (denoted as "2" in Figure 4). Then, since basic block 413c2a has three predecessors, and the required object is not in the $\text{AVAIL}_{\text{OUT}}$ sets for *all* of the preceding basic blocks, it is not added to the $\text{AVAIL}_{\text{IN}}$ set (denoted as "3" in Figure 4). The second pass over the code looks for uses of objects. It checks to make sure that the class pointer that is referenced is in fact in the $\text{AVAIL}_{\text{IN}}$ set of the containing basic block. In the case of the code in Figure 3, the statement `%47 = call i32 %46()` makes use of an object that is not contained in the $\text{AVAIL}_{\text{IN}}$ set (denoted as ERROR in Figure 4). Since, there exists a control flow path where the object is not available, a use-after-free condition exists at this program point.

| Project | Injected Vulns | Discovered Vulns |
|---|---|---|
| Leanify | 1 | 1 |
| libwebm | 2 | 2 |
| lifespan | 1 | 1 |
| MAPIEx | 2 | 2 |
| MonaServer | 2 | 2 |
| 557-animator | 2 | 2 |
| BarsWF | 1 | 1 |
| easyrtc_ie_plugin | 1 | 1 |
| ElasticTabstopsForScintilla | 1 | 1 |

TABLE I: Github projects with injected vulnerabilities

In addition to the code in Figure 1, we tested six other test cases involving use-after-free with various control structures. These tested if-else statements, switch statements, goto statements, and switches combined with if-else and goto, and a switch with an "accidental fall through."

## *B. Manually Generated Vulnerabilities*

To further test the detection capabilities of Available Object Definition Analysis, it was tested on a series of publicly available projects. To find suitable projects, we used a script to identify projects on Github that included Visual Studio project files and made use of heap-allocated objects. Of the first 1000 projects found on Github, 66 met both of those conditions. We selected the first nine projects that would build with minimal modification and injected vulnerabilities into the source. In our corresponding technical report [**?**], we provide the `diff` output from comparing the modified files with the files from the git repositories. With this information, the vulnerable conditions can be easily reproduced.

The vulnerabilities were injected using one of two methods. The first is by freeing objects that had been allocated by the original author. By using objects that were defined by the original author, and only injecting deletions of those objects, we encountered several real-world scenarios that were not seen in our control set tests from Section VI-A. For example, the Leanify [1] project makes use of the factory design pattern and virtual destructors. As described in Section V-C, these conditions make detection of use-after-free conditions more difficult. The second method for injecting use-after-free conditions is by deleting return statements that would cause the use of a freed object to be unreachable.

After injecting the vulnerabilities into the nine test projects, they were each run through AODA. The results of those tests can be seen in Table I. In short, AODA was able to detect every one of the thirteen use-after-free conditions that were injected into the nine Github projects without a single instance of a false positive.

[1]https://github.com/JayXon/Leanify

| Library | Potential Use-after-free Conditions |
|---|---|
| ActionCenterCPL.dll | 5 |
| AdmTmpl.dll | 16 |
| bidispl.dll | 3 |
| cabview.dll | 12 |
| certcli.dll | 1 |
| cewmdm.dll | 8 |
| cnvfat.dll | 1 |
| comdlg32.dll | 1 |
| comsnap.dll | 9 |
| credui.dll | 6 |
| cscapi.dll | 2 |
| dataclen.dll | 2 |
| devenum.dll | 3 |
| devmgr.dll | 2 |
| eapp3hst.dll | 2 |
| eappcfg.dll | 4 |
| Faultrep.dll | 2 |
| FirewallAPI.dll | 1 |
| fontext.dll | 5 |
| fundisc.dll | 4 |
| gpprnext.dll | 1 |
| gpedit.dll | 3 |
| iasrad.dll | 1 |
| icsigd.dll | 1 |
| imagehlp.dll | 1 |
| ipsmsnap.dll | 23 |
| itss.dll | 8 |

TABLE II: DLLs containing use-after-free vulnerabilities

### C. Real-World Vulnerability

In addition to verifying AODA with simple examples, we also confirm that AODA can detect previously-disclosed vulnerabilities. In particular, we analyze a vulnerability in Microsoft Internet Explorer (located in tiptsf.dll) that was disclosed as MS13-069 and CVE-2013-3205. In Figure 5, we can see the AVAIL sets as they were generated during analysis (some basic blocks omitted). Later we see a use of an object of type `??0CEditSession@@QAE@P6GJKPAV0@@Z@Z` that is not in the $\text{AVAIL}_{IN}$ set of the basic block containing the use point. This indicates the presence of a use-after-free vulnerability.

The preceding example represents an interesting case that was caught by the automated analysis. In this case, there was a use of a class pointer before it was ever initialized. While this is not strictly use-after-free vulnerability in the textbook sense[2], the analysis was able to detect it. This is because there was a program point where class type was used, but not in the $\text{AVAIL}_{\text{IN}}$ set for the containing basic block. Since there is a code path in which the class pointer was not instantiated, it was not included in the $\text{AVAIL}_{\text{IN}}$ set.

### D. System32 Directory Results

To estimate the extent of use-after-free vulnerabilities in production operating systems, we ran AODA on 652 binaries found in the system32 directory on a default install of Windows 7. The analysis revealed 127 potential use-after-free conditions. Table II shows the results of the scan of the binaries. These libraries are often used by popular applications (and are thus popular targets for attackers) like Microsoft Office and Internet Explorer.

[2]From the perspective of an adversary, this is exactly the same - a portion of improperly allocated memory is accessed by the program.

## VII. Discussion

In Section VI we validated AODA against simple test cases and a previously disclosed vulnerability, and discovered 127 potential use-after-free conditions in 27 binaries in the system32 directory of a default install of Windows 7. In this section, we address the quality of the results of AODA. As with any static code analysis, the analyses presented in this paper may result in false positives or negatives, and even true positives may not necessarily be exploitable.

### A. False Positives

False positives are always an important concern in static analysis techniques.

AODA only identifies code paths where an object is not guaranteed to be available at a given program point. Like all static analysis, AODA cannot guarantee that a given code path will execute at runtime. Some use-after-free paths may be highly unlikely to execute, or may even have application-specific logic to prevent the use-after-free execution path. In the case of use-after-free conditions, *these are not false positives*. While there may not be any conditions under which the affected code path may be executed, the presence of the deletion of an object that is later needed is bad development practice and should be refactored.

Philosophically, this is in direct contrast to a buffer overflow enabled by the improper use of a library such as `strcpy`. Because it is technically possible to use `strcpy` in a secure fashion, its very presence does not indicate a weakness. However, the static detection of a freed object being used later in a code path cannot be "used properly" and should *never* be allowed to remain in a codebase regardless of the developer's certainty of its execution.

Nevertheless, actual false positives are possible. One example (unrelated to our data flow analysis) is if a developer declares a structure on the stack with the exact layout of a C++ object, RECALL may confuse accesses to that structure as accesses to a C++ object and indicate a use-after-free condition. This extremely rare occurrence could be mitigated by employing techniques like those described by Dolan-Gavitt et al. in [10].

While we have argued philosophically that false positives are not a problem, our results in Sections VI-B and VI-D experimentally confirm this intuition. Specifically, zero false positives were encountered after manually injecting vulnerabilities into nine open source projects, and of the 652 close source binaries scanned, only 27 had any reported use-after-free conditions — indicating that *625 binaries scanned had no false positives*. Additionally, *not one* of the 20 randomly selected warnings that were manually inspected was a false positive. This further validates our claim that AODA is a technique with low false positives.

### B. False Negatives

False negatives are also likely to be rare with AODA, but there are several potential sources.

The first issue is that IDA Pro could fail to correctly recover the binary code. Given the significant improvement in code recovery by IDA Pro in the past decade, this issue is not likely to be common. A second issue is related: Because RECALL object use detection is focused solely on detecting indirect function calls, we will not detect a use-after-free of an object with no virtual functions. If that object has no virtual functions (and hence, no vtable), exploitability is limited to an attacker being able to change member data of a deleted object, not to inject code to be executed by an improper vtable call. While we may miss those vulnerabilities, they are far less common and are arguably less dangerous than ones that permit arbitrary remote code execution.

Other sources of false negatives are fundamental problems with all static analysis. One source of false negatives is if an object is inserted into a collection (like a list or map), the RECALL framework cannot track the object. This is a widely known open problem in static analysis. Another source is that if the control flow graph fails to model program behavior, we may miss use-after-free conditions. One example of this is if a shared object is freed in one thread but then used in another. This is a fundamental issue with static analysis and is not limited to AODA. A final source of false negatives can occur if a library being analyzed exposes objects that linked code later frees. This is simply a result of the analysis not being able to make claims about behavior outside the code being analyzed. Such code exists in violation of best practices that recommend internal objects should not be accessible outside of a library.

### C. Exploitability

Not all use-after-free conditions are exploitable. Certain conditions favorable to the attacker must exist in order to turn these code flaws into actual exploitable vulnerabilities. Specifically, *all* of the following conditions must be met for any of the use-after-free conditions identified in this work to be exploitable:

1) The code path must be reachable at runtime. Determining the reachability of a given code path has been researched extensively with many techniques achieving reasonable levels of success [15], [31], [33]. It is outside the scope of this paper to provide the details on those analyses.
2) The attacker must be able to overwrite the data at the memory location that contained the object prior to it being freed. Since that memory location will be treated as a class pointer at runtime, the attacker must be able to create a data structure that would be interpretd as an object and achieve their goals for exploitation (as described in Section III-B).
3) The object must have some property that is beneficial to an attacker. In the simplest case, an object that contains a vtable will allow an attacker to substitute a vtable pointer to their own data. In more complex scenarios, the ability to overwrite a property of an object may cause the program to function in a way that is beneficial to the attacker. It is arguable that this condition can only be determined manually.

The purpose of the analysis presented in the paper is not to be an automated "vulnerability detector." Rather, it is designed to identify locations in code that represent potential use-after-free conditions. This could be considered similar to what would constitute a compiler warning rather than a compiler error. The warnings that are returned from the AODA analysis may be used to identify vulnerabilities, but further manual analysis is required to ensure all of the requirements listed above could be met. This requirement does not make our contributions trivial, but emphasizes our improvement over current techniques: we only require manual *confirmation*, where before manual *discovery* was required. Drawing again from the compiler warning analogy, production quality code should be free of all errors *and warnings*. As it pertains to vulnerability analysis, AODA allows vulnerability researchers to more rapily identify potentially exploitable bugs, and determine whether Requirement 3 from the list above has been met.

## VIII. CONCLUSIONS

Use-after-free vulnerabilities are an increasingly important class of software vulnerability, yet compilers are incapable of detecting even trivial instances of this error. In this paper, we present the Available Object Definition Analysis data flow algorithm to statically detect use-after-free vulnerabilities. Based on available expression analysis, AODA detects use-after-free conditions by determining if any code path preceding an access to an object contains a deletion of that object. Because analysis of compiled binaries plays a significant role in securing closed-source software, we implemented AODA for the RECALL binary analysis framework. We showed that AODA was able to detect a previously disclosed vulnerability, and we used AODA to identify 127 previously unknown use-after-free conditions in 27 binaries in a default install of Windows 7. Available object definition analysis provides a significant improvement over the state of the art in detecting this dangerous and common vulnerability.

## ACKNOWLEDGEMENTS

This work was supported in part by the US National Science Foundation under grant numbers CNS-1464087 and CNS-1464088. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## REFERENCES

[1] 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.
[2] 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.
[3] J. Caballero, G. Greico, M. Marron, and A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143, 2012.
[4] 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.
[5] Clang. http://clang.llvm.org/.
[6] 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.

[7] 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.

[8] D. Dewey and J. Giffin. Static detection of vtable escape vulnerabilities in binary code. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2011.

[9] 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.

[10] 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.

[11] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest*, 2009.

[12] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan/Feb 2002.

[13] S. Heelan. Vulnerability detection systems: Think cyborg, not robot. *IEEE Security & Privacy*, 9(3):74–77, May-June 2011.

[14] Hey-Rays. http://www.hex-rays.com/.

[15] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. *ACM SIGSOFT Software Engineering Notes*, 25(5):14–25, 2000.

[16] S. Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, 1977.

[17] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[18] 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.

[19] Microsoft. Microsoft Security Bulletin MS13-038. Microsoft TechNet, May 2013. http://technet.microsoft.com/en-us/security/bulletin/ms13-038.

[20] Microsoft. Microsoft Security Bulletin MS13-055. Microsoft TechNet, May 2013. http://technet.microsoft.com/en-us/security/bulletin/ms13-055.

[21] Microsoft. Microsoft Security Bulletin MS13-080. Microsoft TechNet, May 2013. http://technet.microsoft.com/en-us/security/bulletin/ms13-080.

[22] H. Pande and B. Ryder. Data-flow-based virtual function resolution. In *Proceedings of the Third International Symposium on Static Analysis (SAS)*, 1996.

[23] H. D. Pande and B. G. Ryder. Static type determination for C++. In *Proceedings of the 6th USENIX C++ Technical Conference*, 1994.

[24] Pete Becker. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf.

[25] 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.

[26] 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.

[27] 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.

[28] Sparse. https://sparse.wiki.kernel.org/index.php/Main_Page.

[29] Splint. http://splint.org/.

[30] C. Tice. Extended Abstract: Improving Function Pointer Security for Virtual Method Dispatches. GNU Tools Cauldron Workshop, 2012.

[31] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 2000–02, 2000.

[32] 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.

[33] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(02):139–156, 2001.