School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

UAFPrediction: A prediction tool for detecting Use-After-Free vulnerabilities in C

by

Tian Cheng Antheny Yu

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted:	October 2017	Student ID:	z3463699
Supervisor:	Dr Yulei Sui	Topic ID:	3806

Abstract

Use-after-free vulnerabilities are one of the hardest bugs to propagate within an application, but most of applications in the real world are written in low-level languages such as C and C++. There have been many cases where simple mistakes by application developers have resulted in memory corruption vulnerabilities such as use-after-free and security exploits targeting their systems.

In this thesis, we propose UAFPrediction, a prediction tool for detecting use-after-free vulnerabilities in C source code. We set up a work flow to automatically process C source files automatically against multiple use-after-free static detectors and utilised their results with a machine learning model. By utilising machine learning techniques such as support vector machine we can provide an accurate prediction whether a C source file contains a use-after-free vulnerability. However, we note several limitations such as high learning curve of static detectors and low rates of detection.

Acknowledgements

For this work, I would like to thank my supervisor, Dr. Yulei Sui for providing such excellent guidance and vision in order for me to complete this thesis. PhD student Hua Yan for giving me great insight and knowledge into this topic, his background knowledge has ensured that I stayed on the right path for this work. Also, I would like to thank Kai Yang Lim, your support throughout this thesis is immeasurable, allowing me to stay motivated on this work throughout this whole time.

Abbreviations

CORG Compiler Research Group

UNSW University of New South Wales

UAF Use-after-free

CFI Control Flow Integrity

CFG Control Flow Graph

CETS Compiler Enforced Temporal Safety

SmPL Semantic Patch Language

SVM Support Vector Machine

CWE Common Weakness Enumeration

List of Figures

Chapter 1

Introduction

Many applications such as networking protocols and system protocols around the world are written in low-languages such as C or C++ which are vulnerable to several types of memory corruption bugs. The drawback of using such languages is that they lack memory safety (or type safety), this requires developers to be more cautious when writing applications with such languages. One obvious solution would be to develop applications in type-safe languages though unfortunately due to the widespread use of C and C++ and the low-level features required for performance critical use-cases it can be unrealistic to make this change.

The severity of memory corruption vulnerabilities can have great impact on a system when exploited and malicious attackers are working hard to take advantage of these vulnerabilities as the potential impact can be detrimental. According to CWE/SANS memory corruption bugs are positioned at one of the top three most dangerous software errors[1]. Many defences have been developed to prevent memory corruption vulnerabilities and improve memory safety. However, there has been limited research into the problem of detecting and preventing use-after-free bugs. Among CVE identifiers of the Google Chrome browser collected between 2011 and 2013 in Table 1.1, use-after-free bugs are observed to have 40 times more bugs than stack overflow bugs and 3 times more than heap overflow bugs. Use-after-free bugs have a more severe impact in

Severity	**Use-after-free**	**Stack Overflow**	**Heap Overflow**	**Others**
Critical	13	0	0	0
High	582	12	107	11
Medium	80	5	98	12
Total	680	17	208	24

Table 1.1: Number of security vulnerabilities in the Google Chrome browser between 2011 and 2013 reported on Common Vulnerabilities and Exposures

comparison to other vulnerabilities, with 88% of use-after-free bugs categorised under critical or high severity, while 51% of heap overflows are categorized as high severity. Use-after-free vulnerabilities have also been utilised as a significant exploitation vector, as demonstrated in recent security contests [2, 3].

Memory corruption attacks such as use-after-free vulnerabilities can be prevented using dynamic or static analysis of code. Dynamic analysis of code (e.g. Valgrind[4], PIN[5], or libdetox[5]) can be used to dynamically insert safety checks into unsafe binaries at runtime. More sophisticated techniques which utilise dynamic analysis can have performance overheads that exceed 100% and are not feasible to be utilised in practice. Static analysis of code such as the SVF tool currently being developed at CORG in UNSW, can utilise inline reference monitors to implement a safety policy. Since this analysis is done by the compiler, static analysis is usually more efficient than dynamic solutions, however static analysis can produce many false positives as safety policies attempt to match as many violations as possible.

Chapter 2

Background

Memory corruption can be broadly classified into two categories, spatial errors and temporal errors. A pointer becomes invalid by going out of the bounds of its pointed object, dereferencing an out-of-bounds pointer causes a spatial error. A pointer pointing to an object that has been deallocated by another pointer is called a dangling pointer, this can cause a temporal error, where a dangling pointer has been dereferenced.

Use-after-free vulnerabilities occurs when an attempt to access memory after it has been freed, which results in a dangling pointer to a block of dead memory, which may later be reallocated or overwritten. While a dangling pointer by itself does not cause memory corruption, but accessing memory through a dangling pointer can result in information leakage or allow an attacker to modify unexpected memory locations, potentially leading to malicious code execution.

Figure 2.1 demonstrates how attackers can utilise a use-after-free vulnerability to perform a VTable hijacking attack. The program consists of three virtual class definitions: `Person` (the base class, lines 1-4), `Boy` (inherits from Person, lines 6-9), `Girl` (inherits from `Person`, lines 11-14). In line 17, two pointers of type `Person` are declared, which are defined as `p1` and `p2`.

```
1  class Person { public: virtual void talk(); };
2
3  class Boy : public Person { public: void talk(); };
4
5  class Boy : public Person { public: void talk(); };
6
7  int main(int argc, char* argv[]){ Person *p1, *p2; ...
8
9      if (input == "boy"){ p1 = new Boy(); } else { p1 = new
           Girl(); }
10
11     p1->talk(); p2 = p1;
12
13     delete p1;
14
15     p2->talk();
16
17     return 1; }
```

Figure 2.1: Demonstration of use-after-free vulnerability being abused.

The pointer `p1` can either be instantiated either as `Boy` or `Girl` (line 20), dependent on the value of input which we will assume in this scenario as input from the user. For this use case, we will assume input is true, `p1` will be instantiated as `Boy` (line 20). When the virtual method `talk` is called (line 23), the particular implementation, which is defined by `Boy`, will be executed. In line 24, the pointer `p2` is assigned to `p1`, creating a shallow copy of `p1`, essentially the pointers `p1` and `p2` are both pointing to the same memory block. On line 26, `p1` is deleted, the destructor of `Person` and `Boy` are called and the spaced allocated to the `Boy` object is marked as free. This space can now be used for future allocation, and depending on the heap allocator the contents is either zeroed or left as is. However, `p2` still points to the location that `p1` was pointing at and is now become a dangling pointer. If `p2` is accessed, for example shown on line 28, where the `talk` method has been invoked, the behaviour of the program is undefined. There are three possible scenarios: (a) the program crashes, if the freed memory has been zeroed; (b) the `talk` method of boy is called, if the memory deallocator has not zeroed out the memory; (c) arbitrary code gets executed, assuming the freed memory has been deliberately reused.

The example we described above is very simple. In practice, however, real world use-after-free vulnerabilities can be complicated to discover. The allocation, propagation, free and dereference operations could all be located in separate functions and modules. Additionally, at run time, the execution of these operations could occur in different threads. Especially for applications with event-driven designs and object-orientated programming paradigms in mind, the difficulty of discovering use-after-free vulnerabilities becomes harder. For example, web browsers need to handle various events from Javascript or Document Object Model, UI applications need to handle user generated events and server-side applications implement event-loops to handle massive client relationships. Developers are prone to make mistakes when implementing object pointer operations and thus leave the door open for dangling pointer problems. However, not all dangling pointers can cause temporal errors in memory, dangling pointers which are not dereferenced within an application are kept in a pacified state and can be considered a false alarm in this scenario as attackers are unable to utilise these types of dangling pointers to hijack the control flow of an application.

In order to abuse dangling pointers which dereference memory in order to achieve a control-flow hijack or information leak, an attack needs to place useful data in the freed memory region where the dangling pointer is pointing too. There are various ways to exploit unsafe dangling pointers, for example, attackers can place a crafted virtual function table pointer in the freed region, when a virtual function in the table is called later (i.e. memory read dereference on unsafe dangling pointers), a control flow hijack is accomplished. As another example, if the attacker places a root-privileged flag for checking the access rights in the freed region, a privilege escalation attack is accomplished. Moreover, if the attacker places corrupted string length metadata in the freed region and the corresponding string is retrieved, an information leakage attack is accomplished.

The malicious intent of unsafe dangling pointers is dependent on whether an attacker can place crafted data where a dangling pointer points. Specifically, an attack needs to place malicious objects where the freed memory region is located (e.g. an extra memory allocation is a popular exploitation technique). For an attack to be successful, an attack will have needed to perform the malicious operations between the free and use calls (using Figure 2.1, between line 26 and 28) because it is the only time window that the freed memory region can be overwritten and exploited. Thus exploitability of a use-after-free vulnerability is a dependent on whether an attacker can gain control between the free and use calls. For example, the Google Chrome security team determines the bounty on bugs taking this into consideration [6].

2.1 Related work

Research into memory-related issues such as invalid memory access, memory leaks and use-after-free vulnerabilities have been studied for many years. The research community has developed numerous methods to defend against memory corruption bugs in C and C++ applications.

2.1.1 Control flow integrity.

Control Flow integrity (CFI) [7, 8, 9, 10] is a concept which ensures that a program executes only the control flows that are part of its original Control flow graph (CFG). CFI can protect software against arbitrary control-flow hijack attacks as CFI guards function pointers to guarantee legitimate control flows. However, CFI though being a strong defence, most implementations of CFI utilise coarse-grained CFI to avoid heavy performance overheads and false positive alarms. Recent research has shown that coarse-grained CFI implementations can be defeated [11, 12, 13] and also some fine-grained CFI schemes are prone to attacks [14, 15]. Use-after-free vulnerabilities can also be abused to corrupt non-control data such as vector length variables, user privilege bits or sandbox enforcing flags in objects [16], all of which are not function pointers, bypassing CFI protection techniques.

2.1.2 Use-after-free detectors.

Development into the detection of use-after-free vulnerabilities can be categorised into using static analysis or dynamic analysis. Static analysis of dangling pointers is difficult, as these pointers require precise "points-to" and reachability analysis across all inter-procedure paths to if the pointer is dereferenced. The main benefit of using static analysis is that no performance overhead is imposed during the execution. GUEB [17] one such tool which utilises static analysis to detect use-after-free vulnerabilities, unfortunately the tool only targets small programs and is prone to false negatives. Coccinelle [18] is a static analysis tool which utilises Coccinelle Semantic Patches written in the Semantic Patch Language(SmPL). These patches are utilised on C source code to match or replace metavariables defined by Coccinelle Semantic Patches. Coccinelle Semantic Patches have been developed to detect Use-after-free vulnerabilities [19]. CBMC [20] is a static analyser which utilises verification of C/C++ programs to detect a variety of bugs such as buffer overflows, pointer safety, exceptions and user-specified assertions. CBMC can be configured to detect the presence of use-after-free vulnerabilities. SVF is another static analysis use-after-free detector currently in development at CORG in UNSW with the aim of utilising static analysis on large scale applications to find use-after-free vulnerabilities.

Development in static use-after free detectors is sparse thus most use-after-free detectors are currently based on runtime dynamic analysis. Compiler enforced temporal safety [21] (CETS) maintains unique identifiers with each allocated object, associates this information with related pointers and checks that the object is still allocated on pointer dereferences. Undangle[22] is another runtime dynamic analysis tool to detect use-after-free vulnerabilities, it associates each return value of memory allocation function a unique value and utilises dynamic taint analysis to track the propagation of these labels. When the memory is deallocated, Undangle checks which memory blocks are still associated with the corresponding label and determines how unsafe the associated dangling pointers are based on the lifetime of the dangling pointers. Unfortunately, when practically adopting these approaches, the higher run-time overhead of performing the additional tracking is very high, making it less suitable for practical adoption.

2.1.3 Use-after-free protection techniques

Use-after-free vulnerabilities can be prevented by keeping track of pointers to each object and carefully updating all the pointers within a program so that they do not point to memory blocks that can be reused. These approaches are very effective against use-after-free exploitation and experience moderate overhead. DangNULL[23] and FreeSentry [24] are 2 implementations which utilise the idea of invalidating pointers when they are freed from memory. However, DangNULL can only track pointers that are embedded in heap objects and cannot invalidate pointers stored in stack or global memory. FreeSentry does not support multi-threaded programs, thus is impractical to use in vulnerable applications such as severs and browsers.

2.1.4 Secure memory allocators.

Secure allocators provide drop in replacements for the standard memory allocator, these systems do not track individual allocations but try to prevent previously allocated objects from ending up at the same address as previously freed objects. Some secure memory allocators include cling [25], DieHard[26] and DieHarder[27]. These systems can effectively detect use-after-free operations and some achieve low performance overheads, however it has been shown that these systems can allow attackers to force reuse of a freed memory region [23]. Custom allocators can help prevent against use-after-free attacks, but some applications rely on an embedded allocator for better memory management. For instance Google Chrome uses `tcmalloc` [28] and Mozilla uses `jemalloc` [29], a secure allocator can only protect these applications if their embedded allocators are disabled (often not feasible in practice).

2.1.5 Memory error detectors.

Valgrind [4] and Purify [30] are popular solutions for detecting memory errors, assisting developers in debugging applications. These tools are designed to be complete (incurring no false negatives) and detect all types of memory corruption vulnerabilities, however these tools impose a very high performance overhead. AddressSanitizer [31] is another popular tool developed to optimise the method of representing and probing the status of allocated memory. In order to provide this optimisation, AddressSanitizer utilises a quarantine zone that prevents reuse of previously freed memory and attackers can easily leverage various techniques to force reallocation of previously freed memory blocks, such as Heap Spraying [32, 33] and Heap Feng Shui [34] to successfully perform a use-after-free.

2.1.6 Bug detection using machine learning.

Machine learning has been previously used to assist program analysis for bug detection, but there has been little progress for memory safety errors. Fault Invariant Classifier [35] generates machine learning models of program properties known to result from errors. Machine learning can be utilised to selectively allow unsoundness of static analysers when it is likely to reduce false alarms reported by the analyser [36]. Most approaches of utilising machine learning to detect bugs has been to assist program analysis of program properties [37, 38] in order to classify and rank faults within programs in order to reduce the amount of false detections.

2.2 Coccinelle

Coccinelle[18] is a static analysis tool which utilises scripts to match or transform C code. Coccinelle was initially used to deal with code collateral evolutions. Changes in a central component may need to be adapted to a new interface or coding style. Coccinelle utilises scripts called Semantic Patches in order to propagate these changes in code.

```
        @@ expression E; @@
        - free(E);
        + OPENSSL_free(E);
```

Figure 2.2: Replacing instances of `free()` to `OPENSSL_free()`

Coccinelle code matching capabilities can also be used to find bugs in software. The domain specific language Semantic Patch Language (SmPL) has added temporal logic and support for meta-variables. Meta-variables abstract the syntax tree including types, expressions, statements and identifiers. This allows users to match a function using a meta-variable, and then refer to the same function later in the semantic patch. For example in Figure 2.2, this semantic patch uses a meta-variable E as a place holder for an arbitrary variable. Coccinelle will process this semantic patch by removing the variable and the call to `free` and inserting a new call to `OPENSSL_free` and re-inserting the variable according to the original expression.

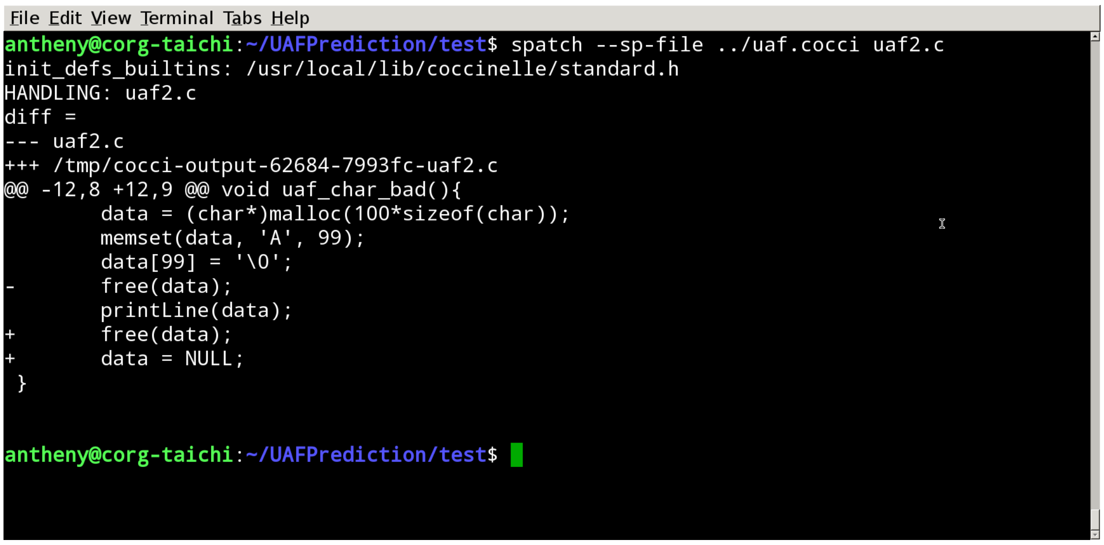
2.2.1 Using Coccinelle to detect use-after-free vulnerabilities

Coccinelle semantic patches can be configured to detect use-after-free memory errors in C/C++ source code. Figure 2.3 describes a semantic patch which can be used in conjunction with Coccinelle to detect UAF bugs. The following command was used to invoke Coccinelle: `spatch --sp-file uaf.cocci file.c`. Where the argument after `--sp-file` is the location of the semantic patch on your system and `file.c` is the C/C++ source file to be examined by Coccinelle.

Figure 2.4 displays the diff file generated by analysing a C/C++ source file with a UAF semantic patch and Coccinelle. This output can be used as a patch on the existing C/C++ source files in order to correct the use-after-free vulnerability.

```
@@
expression E;
expression E2 != NULL;
expression f;
@@
- free(E);
... when != free(E)
(
free(E);
|
E2 = E;
+ free(E);
+ E = NULL;
|
f (<+...E...+>);
+ free(E);
+ E = NULL;
)
```

Figure 2.3: uaf.cocci - A semantic patch used to transform use-after-free bugs



```
antheny@corg-taichi:~/UAFPrediction/test$ spatch --sp-file ../uaf.cocci uaf2.c
init_defs_builtins: /usr/local/lib/coccinelle/standard.h
HANDLING: uaf2.c
diff =
--- uaf2.c
+++ /tmp/cocci-output-62684-7993fc-uaf2.c
@@ -12,8 +12,9 @@ void uaf_char_bad(){
        data = (char*)malloc(100*sizeof(char));
        memset(data, 'A', 99);
        data[99] = '\0';
-       free(data);
        printLine(data);
+       free(data);
+       data = NULL;
 }


antheny@corg-taichi:~/UAFPrediction/test$
```

Figure 2.4: Running Coccinelle with uaf.cocci semantic patch

2.3 CBMC

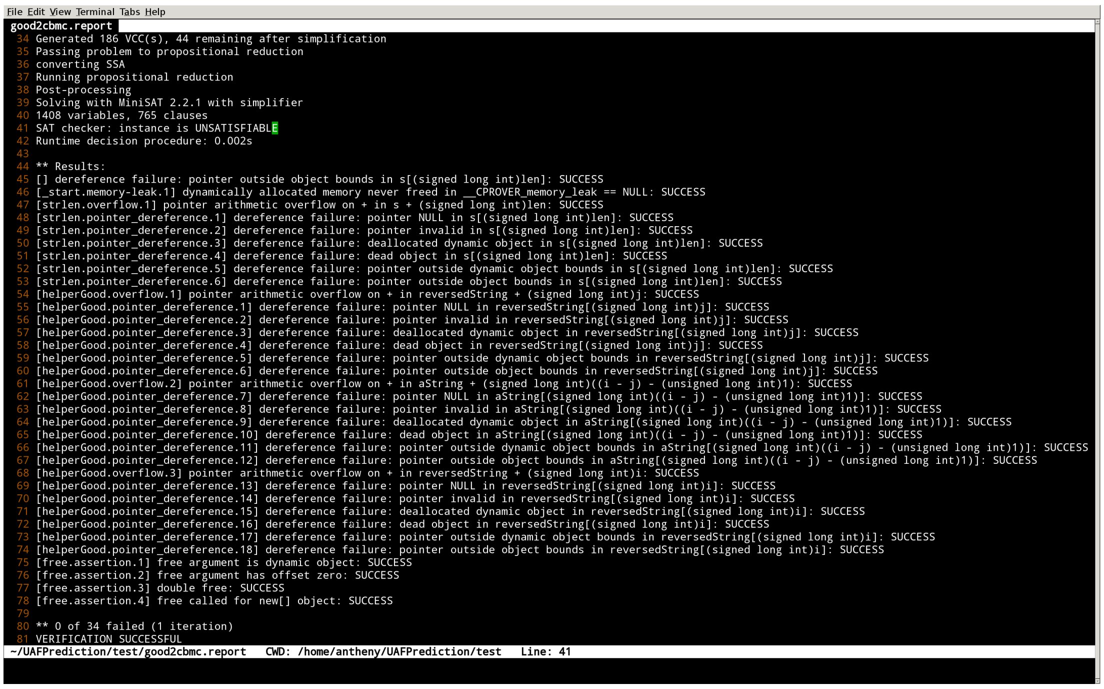
CBMC [20] is a verification tool for C/C++ programs. CBMC uses static analysis to verify array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. CBMC utilises a technique called "Bounded model checking", in combination with a transition relation for a complex state machine and a program's specification, "Bound model checking" obtains a boolean formula by jointly unwinding these two states.

2.3.1 Using CBMC to detect use-after-free vulerablities

CBMC can be used to detect the presence of temporal memory errors such as use-after-free in C/C++ source code. The command to enable CBMC to detect use-after-free bugs is:

```
cbmc --memory-leak-check --pointer-overflow-check --pointner-check
                    --bounds-check --unwind 100 file.c
```

where `--memory-leak-check`, `--pointer-overflow-check`,`--pointner-check` and `--bounds-check` are the application parameters to tell CBMC to check the following properties when analysing a C/C++ source file. The argument `--unwind 100` informs CBMC to only unwind loops 100 times if there is an undefined bound and the argument `file.c` is the input C/C++ source file. Figure 2.5 displays the output of CBMC when it has analysed a simple C program that did not contain a use-after-free bug. From this output, CBMC has performed analysis on various program features and utilises bound model checking to verify if these program features have violated a program's specification.



```
Generated 186 VCC(s), 44 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
1408 variables, 765 clauses
SAT checker: instance is UNSATISFIABLE
Runtime decision procedure: 0.002s

** Results:
[] dereference failure: pointer outside object bounds in s[(signed long int)len]: SUCCESS
[_start.memory-leak.1] dynamically allocated memory never freed in __CPROVER_memory_leak == NULL: SUCCESS
[strlen.overflow.1] pointer arithmetic overflow on + in s + (signed long int)len: SUCCESS
[strlen.pointer_dereference.1] dereference failure: pointer NULL in s[(signed long int)len]: SUCCESS
[strlen.pointer_dereference.2] dereference failure: pointer invalid in s[(signed long int)len]: SUCCESS
[strlen.pointer_dereference.3] dereference failure: deallocated dynamic object in s[(signed long int)len]: SUCCESS
[strlen.pointer_dereference.4] dereference failure: dead object in s[(signed long int)len]: SUCCESS
[strlen.pointer_dereference.5] dereference failure: pointer outside dynamic object bounds in s[(signed long int)len]: SUCCESS
[strlen.pointer_dereference.6] dereference failure: pointer outside object bounds in s[(signed long int)len]: SUCCESS
[helperGood.overflow.1] pointer arithmetic overflow on + in reversedString + (signed long int)j: SUCCESS
[helperGood.pointer_dereference.1] dereference failure: pointer NULL in reversedString[(signed long int)j]: SUCCESS
[helperGood.pointer_dereference.2] dereference failure: pointer invalid in reversedString[(signed long int)j]: SUCCESS
[helperGood.pointer_dereference.3] dereference failure: deallocated dynamic object in reversedString[(signed long int)j]: SUCCESS
[helperGood.pointer_dereference.4] dereference failure: dead object in reversedString[(signed long int)j]: SUCCESS
[helperGood.pointer_dereference.5] dereference failure: pointer outside dynamic object bounds in reversedString[(signed long int)j]: SUCCESS
[helperGood.pointer_dereference.6] dereference failure: pointer outside object bounds in reversedString[(signed long int)j]: SUCCESS
[helperGood.overflow.2] pointer arithmetic overflow on + in aString + (signed long int)((i - j) - (unsigned long int)1): SUCCESS
[helperGood.pointer_dereference.7] dereference failure: pointer NULL in aString[(signed long int)((i - j) - (unsigned long int)1)]: SUCCESS
[helperGood.pointer_dereference.8] dereference failure: pointer invalid in aString[(signed long int)((i - j) - (unsigned long int)1)]: SUCCESS
[helperGood.pointer_dereference.9] dereference failure: deallocated dynamic object in aString[(signed long int)((i - j) - (unsigned long int)1)]: SUCCESS
[helperGood.pointer_dereference.10] dereference failure: dead object in aString[(signed long int)((i - j) - (unsigned long int)1)]: SUCCESS
[helperGood.pointer_dereference.11] dereference failure: pointer outside dynamic object bounds in aString[(signed long int)((i - j) - (unsigned long int)1)]: SUCCESS
[helperGood.pointer_dereference.12] dereference failure: pointer outside object bounds in aString[(signed long int)((i - j) - (unsigned long int)1)]: SUCCESS
[helperGood.overflow.3] pointer arithmetic overflow on + in reversedString + (signed long int)i: SUCCESS
[helperGood.pointer_dereference.13] dereference failure: pointer NULL in reversedString[(signed long int)i]: SUCCESS
[helperGood.pointer_dereference.14] dereference failure: pointer invalid in reversedString[(signed long int)i]: SUCCESS
[helperGood.pointer_dereference.15] dereference failure: deallocated dynamic object in reversedString[(signed long int)i]: SUCCESS
[helperGood.pointer_dereference.16] dereference failure: dead object in reversedString[(signed long int)i]: SUCCESS
[helperGood.pointer_dereference.17] dereference failure: pointer outside dynamic object bounds in reversedString[(signed long int)i]: SUCCESS
[helperGood.pointer_dereference.18] dereference failure: pointer outside object bounds in reversedString[(signed long int)i]: SUCCESS
[free.assertion.1] free argument is dynamic object: SUCCESS
[free.assertion.2] free argument has offset zero: SUCCESS
[free.assertion.3] double free: SUCCESS
[free.assertion.4] free called for new[] object: SUCCESS

** 0 of 34 failed (1 iteration)
VERIFICATION SUCCESSFUL
~/UAFPrediction/test/good2cbmc.report   CWD: /home/antheny/UAFPrediction/test   Line: 41
```

Figure 2.5: The output of CBMC analysing a simple C program with no detected use-after-free vulnerability

2.4 SVF

SVF is a use-after-free detector using static analysis to detect the presence of use-after-free vulnerabilities. SVF can analyse various small to medium scale open source applications developed in C and C++. SVF analyses bitcode files which are generated when C/C++ source code is compiled with the LLVM clang compiler and generates a concise report of each pointer dereference when the associated pointer has been deallocated.

2.4.1 Using SVF to detect use-after-free vulnerabilities

To enable SVF to analyse C/C++ source code to detect use-after-free bugs, it is necessary to generate bitcode files using the LLVM clang compiler. Each C/C++ application needs to be compiled with the following flags `-g -flto` to ensure that the SVF tool is able to evaluate the applications to detect use-after-free vulnerability. For example, to compile a C source file named `file.c` the command will be:

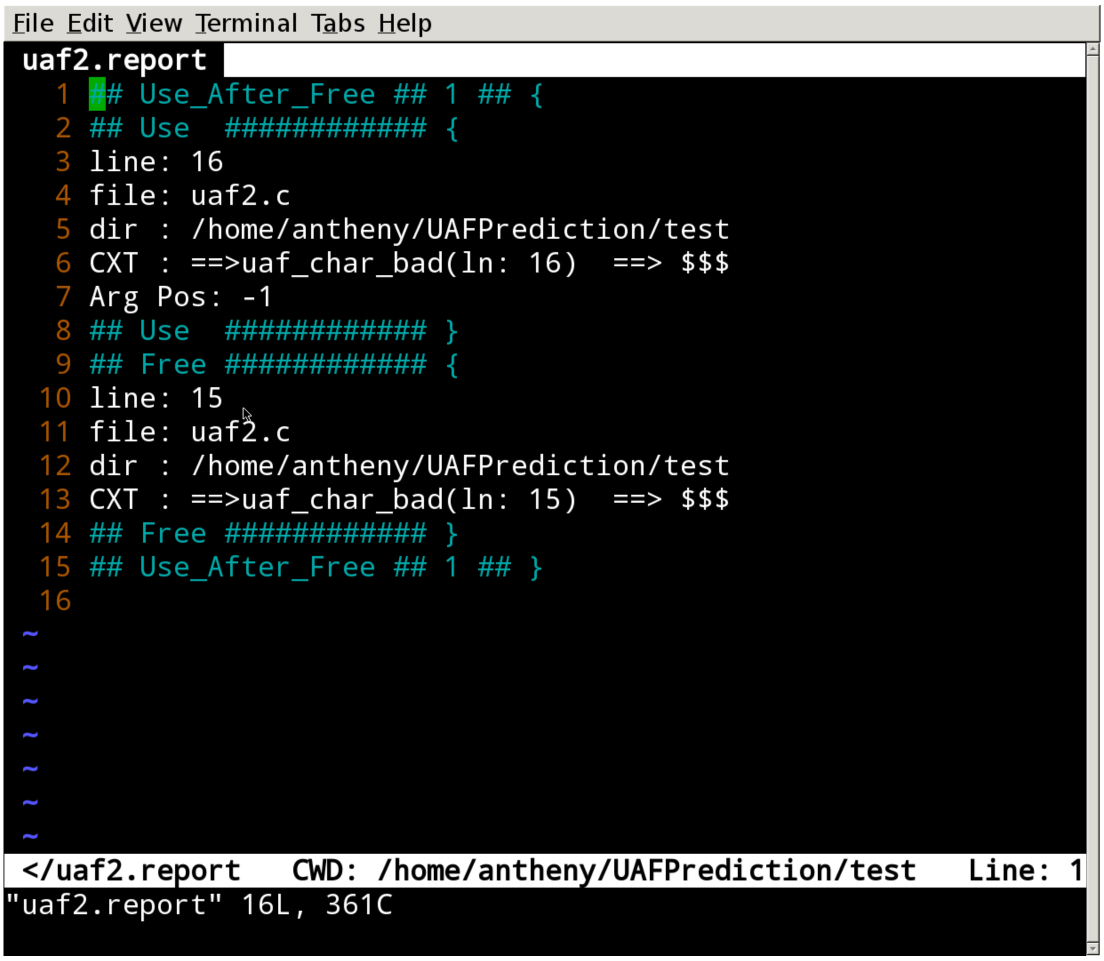
```
clang -g -flto -o file file.c
```

Next, for SVF to analyse a bitcode file, the following command was used:

```
stc -uaf -flowbg=300000 -cxtbg=300000 -pathbg=300000 -stccxt=100
                -singleVFG -dbg=false -mb file.bc
```

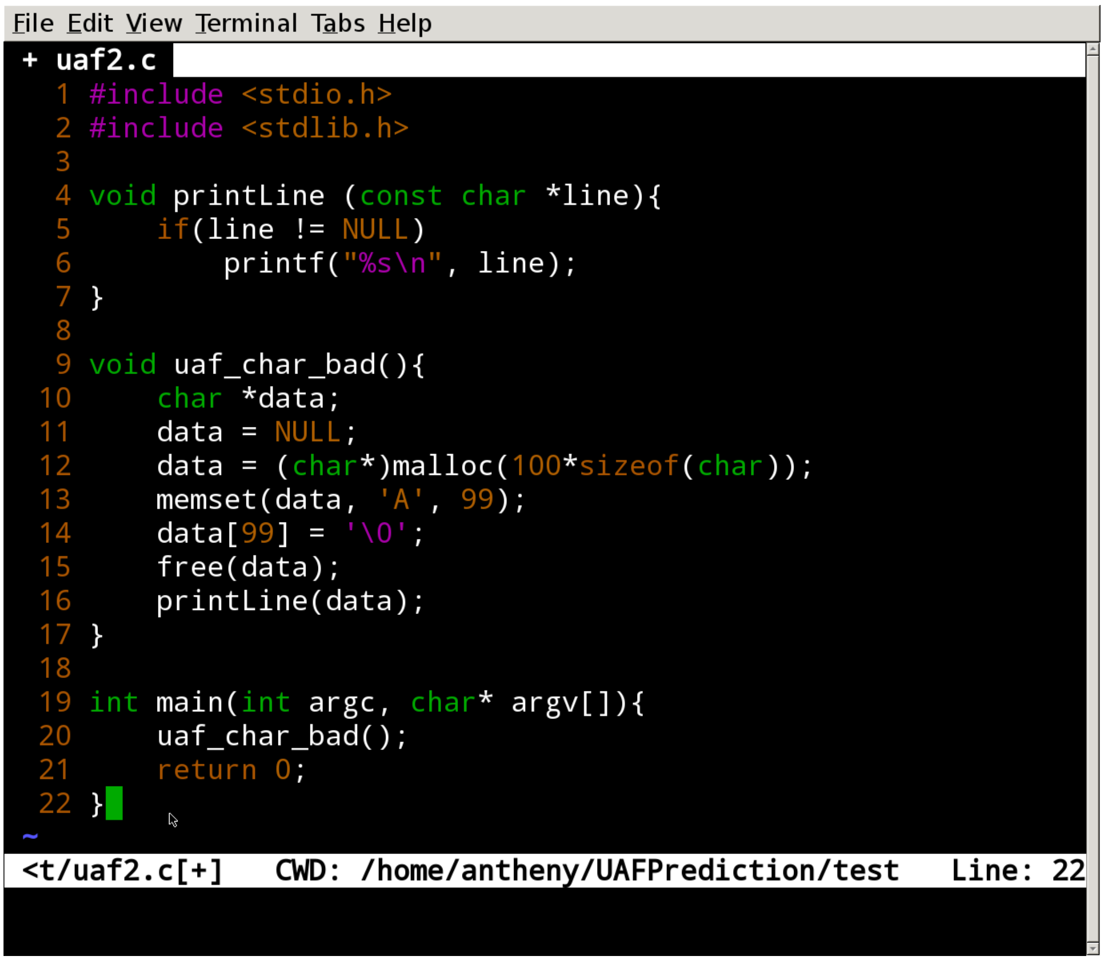
where `flowbg`, `cxtbg`, `pathbg` and `stccxt` are application parameters set by the user to specify the precision and cost tradeoff. After the SVF tool has analysed the bitcode file, the tool generates a report which details use-after-free vulnerabilities.

Figure 2.6 displays the report generated when the SVF tool has analysed the bitcode file of uaf2.c 2.7 as simple C source file with a use-after-free vulnerability. Each use-after-free vulnerability detected by the SVF is characterised by a number to differentiate between the vulnerabilities, with each bug split between the memory dereference (use) and the memory deallocation (free). From Figure 2.6 the SVF report describes the first use-after-free vulnerability under the heading `## Use_After_Free ## 1 ##`. The memory dereference is bordered between line 2 to 8 and concisely displays where the pointer dereference (use) was executed. In this scenario, the malicious command was executed on line 16 within the file `uaf2.c` and also describes the control flow which led to the memory dereference. The memory deallocation (free) that was detected on the object is described between lines 9-14, in this case the associated object had been freed earlier on line 15 in `uaf2.c`



```
uaf2.report
## Use_After_Free ## 1 ## {
## Use  ############ {
line: 16
file: uaf2.c
dir : /home/antheny/UAFPrediction/test
CXT : ==>uaf_char_bad(ln: 16)  ==> $$$
Arg Pos: -1
## Use  ############ }
## Free ############ {
line: 15
file: uaf2.c
dir : /home/antheny/UAFPrediction/test
CXT : ==>uaf_char_bad(ln: 15)  ==> $$$
## Free ############ }
## Use_After_Free ## 1 ## }

~
~
~
~
~
~
~
</uaf2.report   CWD: /home/antheny/UAFPrediction/test   Line: 1
"uaf2.report" 16L, 361C
```

Figure 2.6: SVF report of UAF vulnerabilities of uaf2.c

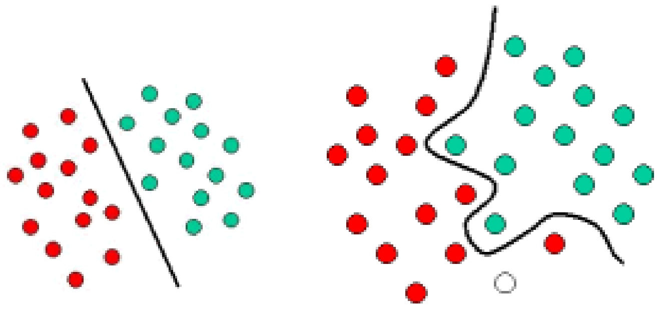


```c
#include <stdio.h>
#include <stdlib.h>

void printLine (const char *line){
    if(line != NULL)
        printf("%s\n", line);
}

void uaf_char_bad(){
    char *data;
    data = NULL;
    data = (char*)malloc(100*sizeof(char));
    memset(data, 'A', 99);
    data[99] = '\0';
    free(data);
    printLine(data);
}

int main(int argc, char* argv[]){
    uaf_char_bad();
    return 0;
}
~
<t/uaf2.c[+]   CWD: /home/antheny/UAFPrediction/test    Line: 22
```

Figure 2.7: uaf2.c C source code

2.5 Support Vector Machine

Support Vector Machines [39] (SVM) are a set of of supervised learning models which utilise learning algorithms to analyse data for classification and regression analysis. Support Vector Machines were established from the idea of decision planes that define decision boundaries. A decision plane is one that separates between a set of objects having different class memberships. Figure 2.8 is an illustration which demonstrates the idea of a decision plane. In this example, the objects belong either to class green or red. The separating line defines a boundary on the right side of which all objects are green and to the left of which all objects are red. Any new object (white circle) falling to the right is classified, as green or classified as red should it fall to the left of



(a) Linear classifier (b) Hyperplane classifier

Figure 2.8: Types of decision planes

the separating line.

Figure 2.8a is typical example of a linear classifier, i.e., a classifier that separates a set of objects into their respective groups (green and red in this example) with a line. However, most classification tasks are not that simple, and often require more complex structures in order to make an optimal separation, i.e., correctly classify new objects on the basis of the trained samples that are available. Figure 2.8b illustrates this scenario, in comparison to figure 2.8a, it is clear that separating the green and red objects into different classes would require a curve (which is more complex than a line). Classification tasks based on drawing separating lines to distinguish between objects of different class memberships are known as hyperplane classifiers. Support Vector Machines are particularly suited to handle such tasks.

Figure 2.9 shows the basic concept underlying Support Vector Machines. Here we see the original objects, shown on the left of the illustration, mapped using a set of mathematical functions, known as kernels. The process of rearranging the objects is known as mapping. Note that in this new setting, the mapped objects, shown on the right of the illustration, is linearly separable and, thus, instead of constructing the complex curve (left illustration), all we have to do is to find an optimal line that can separate the green and the red samples.

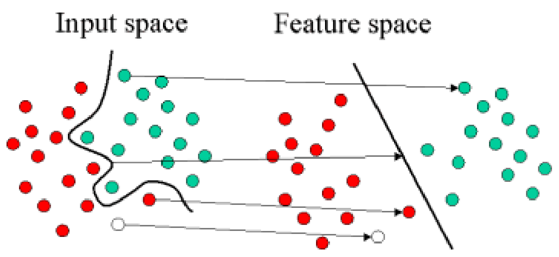


Figure 2.9: Transformation of hyperplane to linear classifier

Chapter 3

My Work: UAFPrediction

UAFPrediction is accessible from `https://github.com/yuleisui/UAFPrediction`.

UAFPrediction is a python script which predicts the presence of use-after-free vulnerabilities in C and C++ source code using a Support Vector Machine (SVM) model. UAFPrediction utilises the outputs of the Coccinelle tool [18] with a UAF Coccci patch, CBMC [20] and SVF [40], then it utilises training samples from the juliet test suite [41] to predict whether there are any use-after-free bugs present in C source code.

The UAFPrediction tool is divided into 2 components, the python wrapper scripts which automate data collection and processing of Coccinelle, CBMC and SVF tools and the front-end command line interface that predicts if there exist use-after-free bugs in C source code based on a pre-trained set of examples supplied from the juliet test suite.

3.1 Python wrapper scripts

The python wrapper scripts are utilised by the UAFPrediction tool to simulate the invocation of Coccinelle, CBMC and SVF from the command line. These scripts mainly serve the purpose of analysing the output made available from running the tools with their appropriate arguments. The python wrapper script also integrates the work flow of these tools, abstracting unnecessary details away from the user to provide an easy learning curve to predicting use-after-free bugs with C source code.

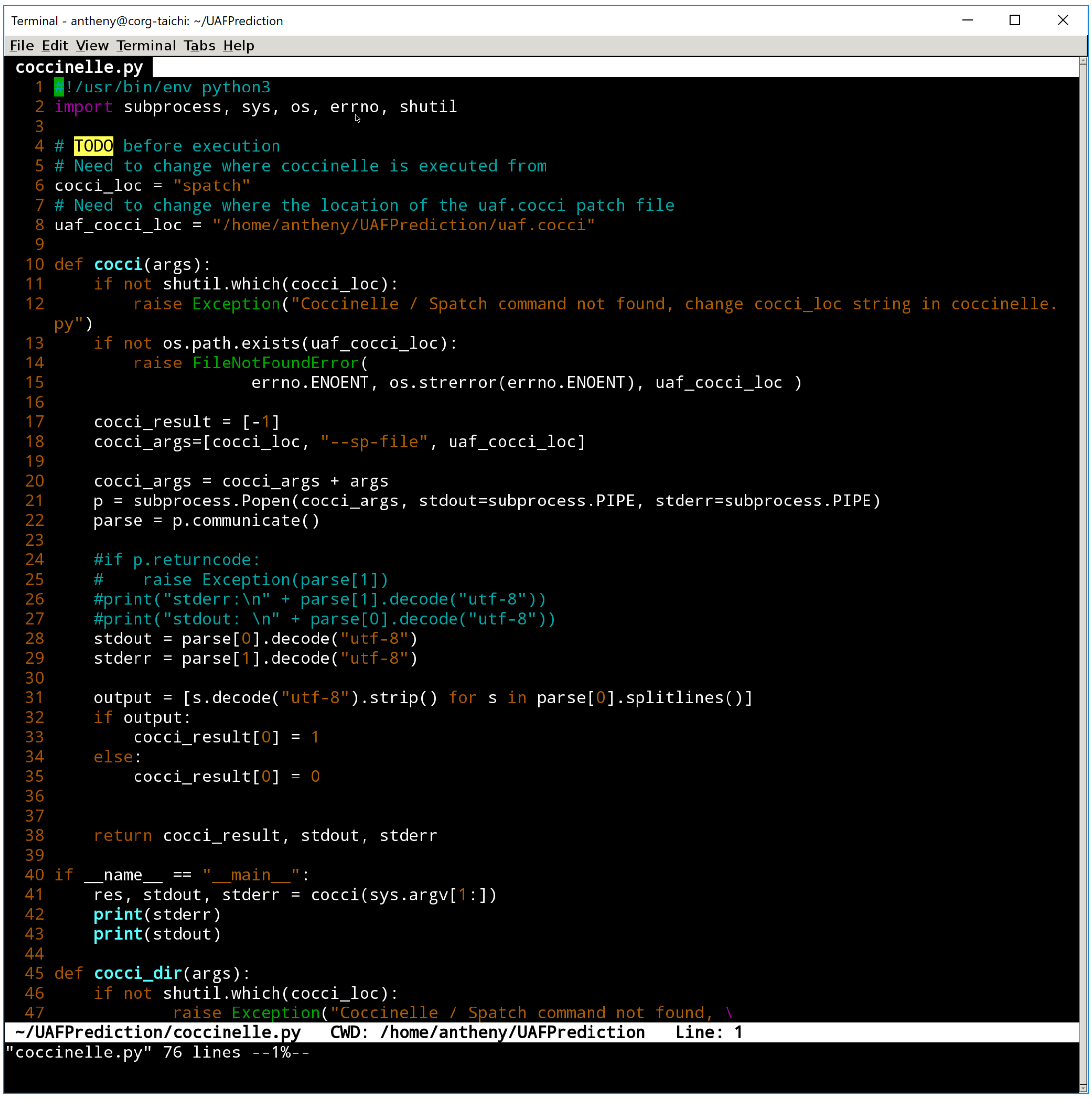
3.1.1 Coccinelle.py

Coccinelle.py is a python script which simulates the coccinelle tool on the command line and utilising uaf.cocci as the semantic patch as described in section 2.2.1. The wrapper script takes in a list of C source files and executes coccinelle with the semantic patch. The resulting analysis reported from coccinelle is a diff patch which displays the transformations imposed to correct the use-after-free vulnerability. The coccinelle.py python script takes this output and uses regular expression to determine whether coccinelle reported a use-after-free vulnerability.

For the UAFPrediction tool to be able to utilise coccinelle to extract information on use-after-free bugs in C source code, the coccinelle.py python script must be configured according to the user's system. Figure 3.1 displays the necessary parts of coccinelle.py that are needed to be modified. On line 6, the variable `cocci_loc` must be modified to where the coccinelle is installed on your system. If coccinelle was installed using a package manager, then it can be invoked by using the "spatch" command, otherwise modify the variable to the location of the binary. On line 8, the variable `uaf_cocci_loc` must be modified to the location of the uaf.cocci semantic patch file.

3.1.2 cbmc.py

The second python wrapper script used by the UAFPrediction tool is cbmc.py, which simulates the CBMC tool on the command line as described in section 2.3.1. The wrapper script takes in a list of C source files and executes CBMC with options that will allow CBMC to report use-after-free vulnerabilities. The resulting output reported by CBMC is a summary of the program properties CBMC verified. The wrapper script, cbmc.py, parses this summary using regular expressions to match any references

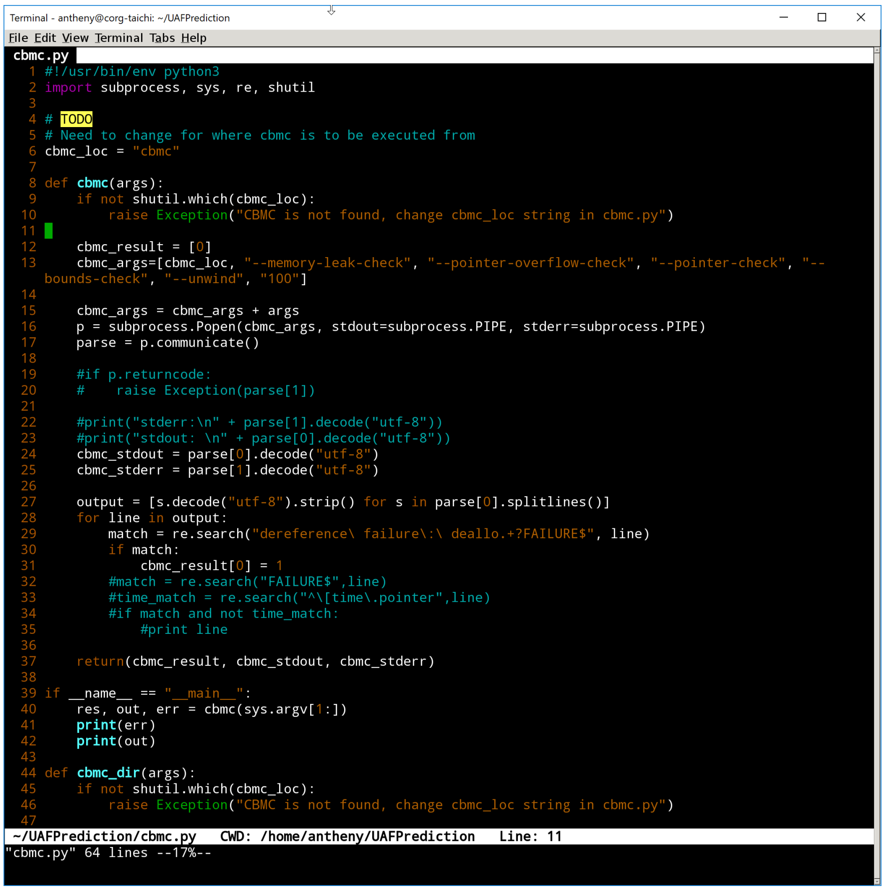

Terminal - antheny@corg-taichi: ~/UAFPrediction
File Edit View Terminal Tabs Help
coccinelle.py

```
#!/usr/bin/env python3
import subprocess, sys, os, errno, shutil

# TODO before execution
# Need to change where coccinelle is executed from
cocci_loc = "spatch"
# Need to change where the location of the uaf.cocci patch file
uaf_cocci_loc = "/home/antheny/UAFPrediction/uaf.cocci"

def cocci(args):
    if not shutil.which(cocci_loc):
        raise Exception("Coccinelle / Spatch command not found, change cocci_loc string in coccinelle.py")
    if not os.path.exists(uaf_cocci_loc):
        raise FileNotFoundError(
                    errno.ENOENT, os.strerror(errno.ENOENT), uaf_cocci_loc )

    cocci_result = [-1]
    cocci_args=[cocci_loc, "--sp-file", uaf_cocci_loc]

    cocci_args = cocci_args + args
    p = subprocess.Popen(cocci_args, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    parse = p.communicate()

    #if p.returncode:
    #    raise Exception(parse[1])
    #print("stderr:\n" + parse[1].decode("utf-8"))
    #print("stdout: \n" + parse[0].decode("utf-8"))
    stdout = parse[0].decode("utf-8")
    stderr = parse[1].decode("utf-8")

    output = [s.decode("utf-8").strip() for s in parse[0].splitlines()]
    if output:
        cocci_result[0] = 1
    else:
        cocci_result[0] = 0


    return cocci_result, stdout, stderr

if __name__ == "__main__":
    res, stdout, stderr = cocci(sys.argv[1:])
    print(stderr)
    print(stdout)

def cocci_dir(args):
    if not shutil.which(cocci_loc):
            raise Exception("Coccinelle / Spatch command not found, \
 ~/UAFPrediction/coccinelle.py   CWD: /home/antheny/UAFPrediction   Line: 1
"coccinelle.py" 76 lines --1%--
```

Figure 3.1: Coccinelle.py: A python wrapper script for the Coccinelle tool to detect use-after-free vulnerabilities

to use-after-free vulnerabilities.

For the UAFPrediction tool to be able to utilise CBMC to extract information, the cbmc.py python script must be configured according to the user's system. Figure 3.2 displays the essential parts of cbmc.py that need to be modified. On line 6, the variable `cbmc_loc` must be changed to where the CBMC tool is invoked on your system. If CBMC was installed using a package manager, then it can be invoked using the "cbmc" command, otherwise modify the variable to where the binary is located on your system.



Terminal - antheny@corg-taichi: ~/UAFPrediction

```
#!/usr/bin/env python3
import subprocess, sys, re, shutil

# TODO
# Need to change for where cbmc is to be executed from
cbmc_loc = "cbmc"

def cbmc(args):
    if not shutil.which(cbmc_loc):
        raise Exception("CBMC is not found, change cbmc_loc string in cbmc.py")

    cbmc_result = [0]
    cbmc_args=[cbmc_loc, "--memory-leak-check", "--pointer-overflow-check", "--pointer-check", "--
bounds-check", "--unwind", "100"]

    cbmc_args = cbmc_args + args
    p = subprocess.Popen(cbmc_args, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    parse = p.communicate()

    #if p.returncode:
    #    raise Exception(parse[1])

    #print("stderr:\n" + parse[1].decode("utf-8"))
    #print("stdout: \n" + parse[0].decode("utf-8"))
    cbmc_stdout = parse[0].decode("utf-8")
    cbmc_stderr = parse[1].decode("utf-8")

    output = [s.decode("utf-8").strip() for s in parse[0].splitlines()]
    for line in output:
        match = re.search("dereference\ failure\:\ deallo.+?FAILURE$", line)
        if match:
            cbmc_result[0] = 1
        #match = re.search("FAILURE$",line)
        #time_match = re.search("^\[time\.pointer",line)
        #if match and not time_match:
            #print line

    return(cbmc_result, cbmc_stdout, cbmc_stderr)

if __name__ == "__main__":
    res, out, err = cbmc(sys.argv[1:])
    print(err)
    print(out)

def cbmc_dir(args):
    if not shutil.which(cbmc_loc):
        raise Exception("CBMC is not found, change cbmc_loc string in cbmc.py")

~/UAFPrediction/cbmc.py   CWD: /home/antheny/UAFPrediction   Line: 11
"cbmc.py" 64 lines --17%--
```

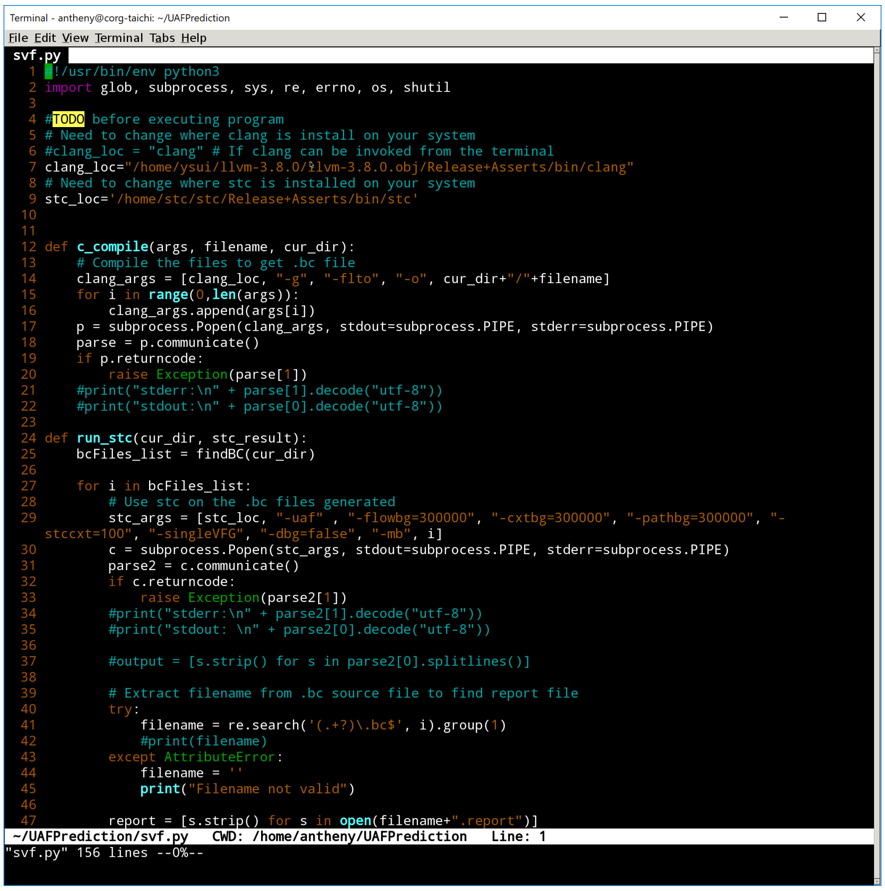
Figure 3.2: cbmc.py: A python wrapper script for the CBMC tool to detect use-after-free vulnerabilities

3.1.3 svf.py

The final python wrapper script used by the UAFPrediction tool is svf.py, which simulates the SVF too on the command line as described in section 2.4.1. The wrapper script takes in a list of C source files and using the LLVM clang compiler, the script will compile the source code to generate bitcode files. Bitcode files are used by the SVF tool to perform pointer analysis to report the presence of use-after-free vulnerabilities. The resulting report generated by SVF is a summary of detected use-after-free bugs, svf.py, parses this summary to detect if SVF reported any warnings.

In order for the UAFPrediction tool to be able to utilise SVF to extract information, the svf.py python script must be configured according to the user's system. Figure 3.3

illustrates the necessary components of svf.py that need to be modified. On line 7, the variable `clang_loc` must be modified to where the LLVM clang compiler is located on your system. If the clang compiler was installed using a package manager, then it can be invoked using the “clang” command, otherwise modify the variable to where the binary is located on your system. Similarly, on line 9 the variable `stc_loc` must be modified to where the SVF binary is located on your system.


Terminal - antheny@corg-taichi: ~/UAFPrediction
File Edit View Terminal Tabs Help
svf.py

```
  1 #!/usr/bin/env python3
  2 import glob, subprocess, sys, re, errno, os, shutil
  3 
  4 #TODO before executing program
  5 # Need to change where clang is install on your system
  6 #clang_loc = "clang" # If clang can be invoked from the terminal
  7 clang_loc="/home/ysui/llvm-3.8.0/llvm-3.8.0.obj/Release+Asserts/bin/clang"
  8 # Need to change where stc is installed on your system
  9 stc_loc='/home/stc/stc/Release+Asserts/bin/stc'
 10 
 11 
 12 def c_compile(args, filename, cur_dir):
 13     # Compile the files to get .bc file
 14     clang_args = [clang_loc, "-g", "-flto", "-o", cur_dir+"/"+filename]
 15     for i in range(0,len(args)):
 16         clang_args.append(args[i])
 17     p = subprocess.Popen(clang_args, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
 18     parse = p.communicate()
 19     if p.returncode:
 20         raise Exception(parse[1])
 21     #print("stderr:\n" + parse[1].decode("utf-8"))
 22     #print("stdout:\n" + parse[0].decode("utf-8"))
 23 
 24 def run_stc(cur_dir, stc_result):
 25     bcFiles_list = findBC(cur_dir)
 26 
 27     for i in bcFiles_list:
 28         # Use stc on the .bc files generated
 29         stc_args = [stc_loc, "-uaf" , "-flowbg=300000", "-cxtbg=300000", "-pathbg=300000", "-
    stccxt=100", "-singleVFG", "-dbg=false", "-mb", i]
 30         c = subprocess.Popen(stc_args, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
 31         parse2 = c.communicate()
 32         if c.returncode:
 33             raise Exception(parse2[1])
 34         #print("stderr:\n" + parse2[1].decode("utf-8"))
 35         #print("stdout: \n" + parse2[0].decode("utf-8"))
 36 
 37         #output = [s.strip() for s in parse2[0].splitlines()]
 38 
 39         # Extract filename from .bc source file to find report file
 40         try:
 41             filename = re.search('(.+?)\.bc$', i).group(1)
 42             #print(filename)
 43         except AttributeError:
 44             filename = ''
 45             print("Filename not valid")
 46 
 47         report = [s.strip() for s in open(filename+".report")]
 ~/UAFPrediction/svf.py   CWD: /home/antheny/UAFPrediction   Line: 1
"svf.py" 156 lines --0%--
```

Figure 3.3: svf.py: A python wrapper script for the SVF tool to detect use-after-free vulnerabilities

3.2 UAFPrediction.py

UAFPrediction.py is the front-end command line interface which utilises the data collected from the python helper scripts and uses a support vector machine model to provide a prediction if there exist use-after-free bugs in C code. A subset of the juliet test suite is used to train the support vector machine.

Support Vector Machine

The UAFPrediction.py script utilises a support vector machine learning model using the python scikit-learn [42] as the machine learning library to train and predict the presence of use-after-free vulnerabilities in C source code.

There are 3 typically used Support Vector Machine classifiers: linear support, nu-support and C-support. UAFPrediction uses a C-support classifier which provides the best balance of prediction accuracy and execution speed. Given the number of trained samples is less than 10000, a linear support classifier is unnecessary as the execution speed is acceptable and a nu-support classifier provided the same results as a C-support classifier.

The matrix used by the SVM model in UAFPrediction consists of 3 features, as illustrated in figure 3.4, whether cbmc detected a UAF bug, whether coccinelle detected a UAF bug and whether SVF detected a UAF bug. UAFPrediction utilises their respective python helper script as described in section 3.1 to determine the contents of the matrix. UAFPrediction will call each corresponding python helper script with the C source files and the helper scripts will run their respective tools with the source files. The python helper scripts will return either 0 or 1 to indicate whether a UAF bug was detected and the tool's associated output to display to the user if a use-after-free bug was detected.

The support vector machine was trained using examples provided by the juliet test suite using the matrix described above. There were over 200 trained samples with an equal distribution of false and true use-after-free vulnerabilities. The parameters used for the C-support classifier were `c=100` and `gamma=0.001`, which were recommended values based on [43].

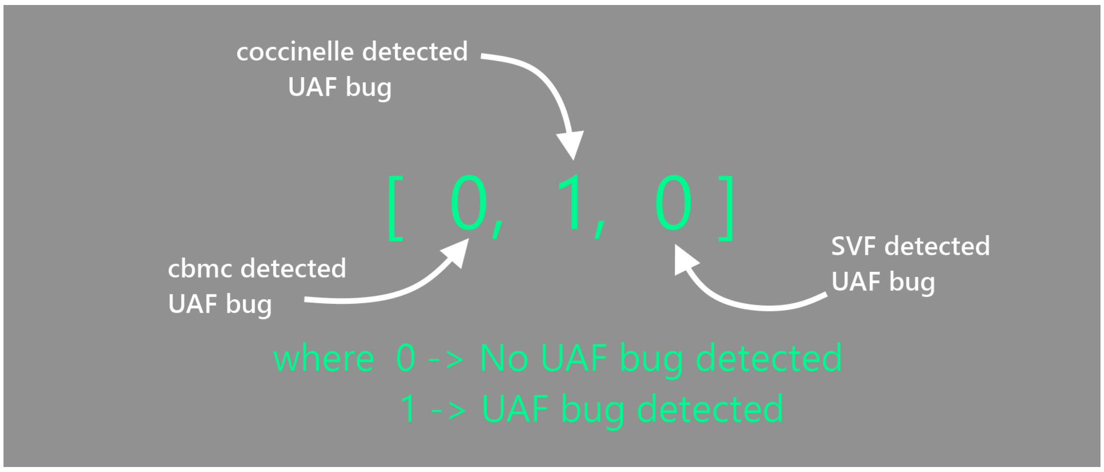
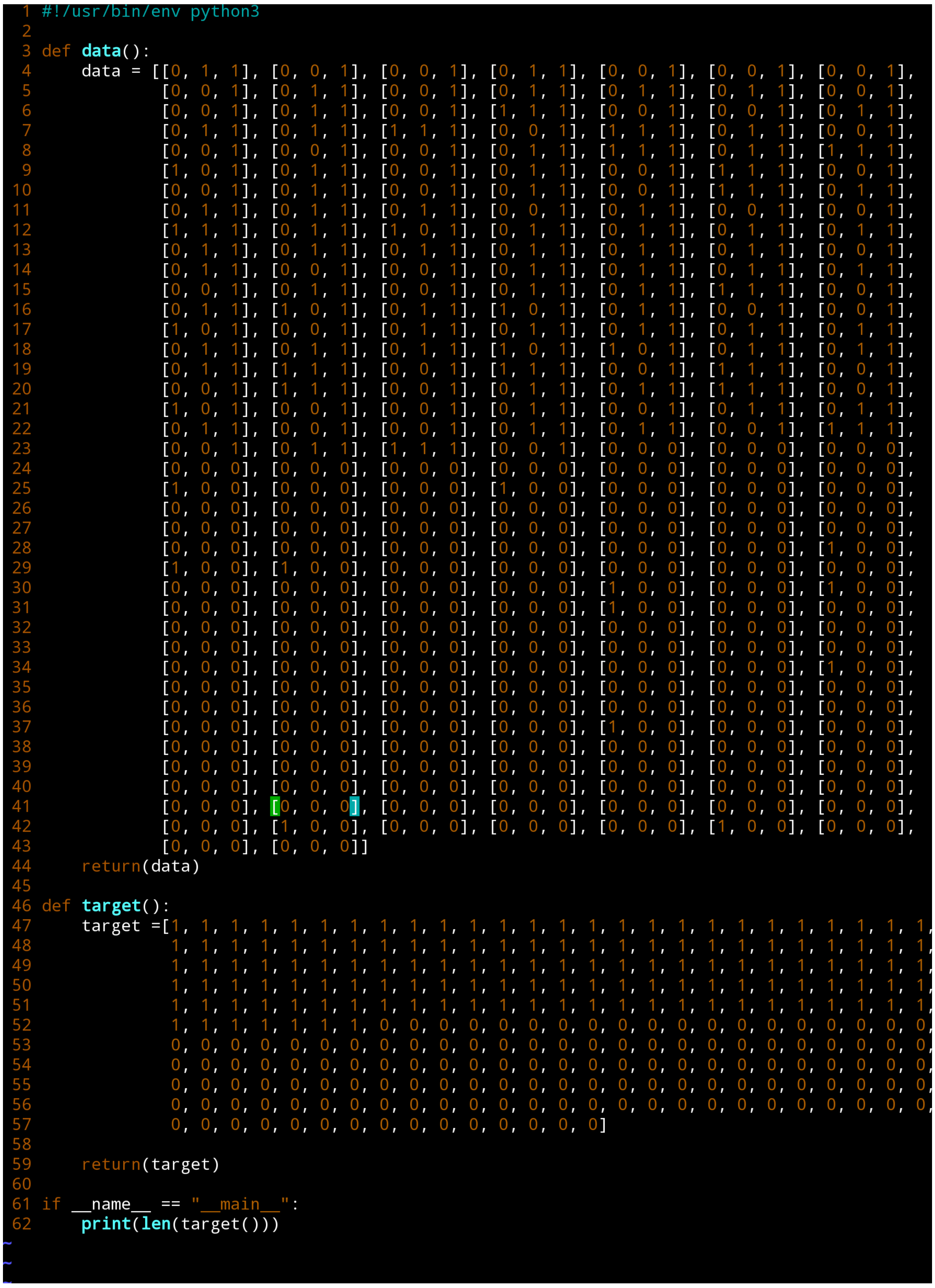


Figure 3.4: The matrix containing the features of the Support Vector Machine

Juliet Test Suite

The juliet test suite [41] is a collection of synthetic C programs with known flaws. These programs can be utilised as test cases to test the effectiveness of static analysers and other software assurance tools. The juliet test suite covers 181 various kinds of flaws documented by common weakness enumeration (CWE). The UAFPrediction tool only utilises the subset of test cases with CWE's related to use-after-free vulnerabilities (CWE416).

The test cases in the juliet test suite provides examples of flaws in differing variations of data structures and different control-flow and data-flow patterns. Each test file contains a bad section containing a vulnerability and a good section which is the correct execution of the test case with no vulnerability. These test cases are split up into their respective good and bad sections and are used as training samples to train the support vector machine as described in section 3.2.

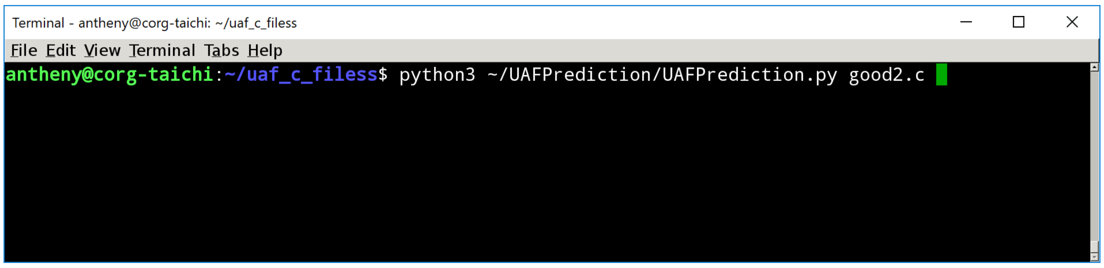


```
#!/usr/bin/env python3

def data():
    data = [[0, 1, 1], [0, 0, 1], [0, 0, 1], [0, 1, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1],
            [0, 0, 1], [0, 1, 1], [0, 0, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 0, 1],
            [0, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1], [0, 0, 1], [0, 0, 1], [0, 1, 1],
            [0, 1, 1], [0, 1, 1], [1, 1, 1], [0, 0, 1], [1, 1, 1], [0, 1, 1], [0, 0, 1],
            [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 1, 1], [1, 1, 1], [0, 1, 1], [1, 1, 1],
            [1, 0, 1], [0, 1, 1], [0, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1], [0, 0, 1],
            [0, 0, 1], [0, 1, 1], [0, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1], [0, 1, 1],
            [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 0, 1], [0, 1, 1], [0, 0, 1], [0, 0, 1],
            [1, 1, 1], [0, 1, 1], [1, 0, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1],
            [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 0, 1],
            [0, 1, 1], [0, 0, 1], [0, 0, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1],
            [0, 0, 1], [0, 1, 1], [0, 0, 1], [0, 1, 1], [0, 1, 1], [1, 1, 1], [0, 0, 1],
            [0, 1, 1], [1, 0, 1], [0, 1, 1], [1, 0, 1], [0, 1, 1], [0, 0, 1], [0, 0, 1],
            [1, 0, 1], [0, 0, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1],
            [0, 1, 1], [0, 1, 1], [0, 1, 1], [1, 0, 1], [1, 0, 1], [0, 1, 1], [0, 1, 1],
            [0, 1, 1], [1, 1, 1], [0, 0, 1], [1, 1, 1], [0, 0, 1], [1, 1, 1], [0, 0, 1],
            [0, 0, 1], [1, 1, 1], [0, 0, 1], [0, 1, 1], [0, 1, 1], [1, 1, 1], [0, 0, 1],
            [1, 0, 1], [0, 0, 1], [0, 0, 1], [0, 1, 1], [0, 0, 1], [0, 1, 1], [0, 1, 1],
            [0, 1, 1], [0, 0, 1], [0, 0, 1], [0, 1, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1],
            [0, 0, 1], [0, 1, 1], [1, 1, 1], [0, 0, 1], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [1, 0, 0], [0, 0, 0], [0, 0, 0], [1, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [1, 0, 0],
            [1, 0, 0], [1, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [1, 0, 0], [0, 0, 0], [1, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [1, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [1, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [1, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
            [0, 0, 0], [1, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [1, 0, 0], [0, 0, 0],
            [0, 0, 0], [0, 0, 0]]
    return(data)

def target():
    target =[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
             1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
             1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
             1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
             1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
             1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

    return(target)

if __name__ == "__main__":
    print(len(target()))
```

Figure 3.5: Screenshot of the test data extracted from a subset of the Juliet Test Suite

Running UAFPrediction

To use UAFPrediction, it must be invoked using python3 from the directory containing the C source code to be analysed. The arguments provided to UAFPrediction are the C source files.


Terminal - antheny@corg-taichi: ~/uaf_c_filess
File Edit View Terminal Tabs Help

```
antheny@corg-taichi:~/uaf_c_filess$ python3 ~/UAFPrediction/UAFPrediction.py good2.c
```

Figure 3.6: Command to execute UAFPrediction

Once UAFPrediction is executed, it will run the python helper scripts automatically with the provided C source files to extract information and create a feature matrix representing whether use-after-free bugs were detected by Coccinelle, CBMC and SVF. Next, it will create a SVM model with a C-Support classifier and utilise test cases from the juliet test suite to train the support vector machine. Finally, the SVM model will take the feature matrix of the C source files and predict if there exist use-after-free bugs. If use-after-free bugs are detected, the tool will report the output from Coccinelle, CBMC and SVF so that the user can investigate further. Otherwise the tool will report "No Use-After-Free bugs have been predicted".

Chapter 4

Evaluation

In this evaluation, we discuss the prediction performance of using a support vector machine to detect use-after-free bugs and the choice of features which effectively predict the presence of use-after-free bugs.

4.1 Prediction performance

Given a limited set of data samples it is important for a prediction model to not use the existing parameters and test it on the same data. A model that just repeats the labels of the samples that it has seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called overfitting. To avoid it, it is widespread practice when utilising machine learning to hold out part of the available data as a test set. However, we reduce the number of samples that can be used for learning the model, thus we can utilise k-fold cross-validation as a procedure to assist in validating the model. In k-fold cross-validation, the training set is split into k smaller sets, then for each of the k "folds", a model is trained using $k-1$ "folds" as the training samples and the resulting model is validated on the remaining "fold" of data.

We evaluate the effectiveness of the prediction performance by utilising 10-fold cross-validation to test the prediction accuracy of the support vector machine. Using the trained samples from the juliet test suite [3.2], we performed a 10-fold cross validation. The mean score with a 10-fold cross-validation was 1 with a 95% confidence interval of (+/- 0.0). This shows the support vector machine has been configured correctly and matches the prediction performance we have been experiencing with C source files that we have manually executed with.

4.1.1 Feature Design

The C-support classifier utilised 3 features in the support vector machine, detecting if Coccinelle, CBMC and SVF report a use-after-free vulnerability. Given the nature of the reports generated by each tool, it was difficult to extract more meaningful features such as pointer information or the call stack reported by pointer dereferences.

4.2 Discussion

The usability of UAFPrediction is hampered by the lack of real-world examples and limitations in the tools to detect use-after-free vulnerabilities.

The CBMC tool produced the lowest rate of detection of use-after-free vulnerabilities given 136 test samples provided by the juliet test suite. Table 4.1 illustrates the number of use-after-free bugs detected by each tool. Given CBMC markets itself as a tool to be used on embedded, it may be difficult to extract meaningful information from CBMC when it is executed on consumer applications which tend to utilise third party libraries. The learning curve to using CBMC is higher than SVF and Coccinelle, with the output verifying every property that exists in an application. Understanding the output of CBMC could potentially waste more time than finding bugs themselves.

The Coccinelle tool with the uaf.cocci semantic patch is effective at simple use-after-free vulnerabilities though the tool struggles detecting vulnerabilities that are inter-procedural. The Coccinelle tool requires users to be familiar with SmPL to create

Use-After-Free detector	Number of UAF bugs detected
Coccinelle	80
CBMC	24
SVF	136

Table 4.1: Detected vulnerabilities by each tool given 136 test files with a Use-After-Free vulnerability from the juliet test suite

effective semantic patches for Coccinelle to match and transform more complex use-after-free characteristics.

The SVF tool has high accuracy in detecting use-after-free vulnerabilities from the juliet test suite, though from my initial analysis of 17 small to medium scale open source C and C++ programs, utilising SVF has its disadvantages. As observed in table 4.2, 4 out of 17 programs produced reports with abnormally large number use-after-free bugs. Currently, utilising static analysis to detect the presence of use-after-free vulnerabilities may produce an excessive number of false positives, making it not ideal for developers to utilise in practice. For the developer, it can be time consuming to analyse each vulnerability to truly check if it is a legitimate use-after-free bug as these can located in separate functions and be executed across different threads.

Application	UAF vulnerabilities reported
Bloaty	0
Bison	1
Cpp-Check-1.79	0
curl	0
ed	0
flex-2.6.0	5
gdb	0
gs	3
gzip	1
keepassx	0
netdata	2
pcre-8.35	48
redis	20
sed	31
uncrustify	0
yaml-cpp-0.5.3	0
zfs	73

Table 4.2: Reported vulnerabilities of using the SVF use-after-free detector on 17 open-source applications developed in C or C++

Chapter 5

Conclusion

In this thesis, we attempted to apply machine learning techniques to create a model to reduce the amount of false positives detected by use-after-free static detectors. The aim to reduce a developer's work flow when a severe use-after-free bug is detected is difficult to achieve. Some tools require a high learning curve to effectively extract meaningful and related information to detect use-after-free bugs, while other tools have a low rate of detection of true use-after-free vulnerabilities. This makes it difficult for a support vector machine models to accurately predict the presence of use-after-free bugs in C source code.

Bibliography

[1] "2011 cwe/sans top 25 most dangerous software errors," 2011. [Online]. Available: http://cwe.mitre.org/top25/

[2] "Adobe flash player as3 convolutionfilter use-after-free remote code execution vulnerability," 2015. [Online]. Available: http://www.zerodayinitiative.com/advisories/ZDI-15-134/

[3] Mozilla, "Use-after-free in typeobject," 2014. [Online]. Available: https://www.mozilla.org/en-US/security/advisories/mfsa2014-30/

[4] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[6] "The google chrome project - issues," 2017. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/list

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 2005, pp. 340–353.

[8] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the 18th ACM conference on Computer and communications security.* ACM, 2011, pp. 29–40.

[9] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 2013, pp. 559–573.

[10] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *Usenix Security*, vol. 13, 2013.

[11] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection." in *USENIX Security*, vol. 14, 2014.

[12] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *USENIX Security*, vol. 14, 2014.

[13] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on.* IEEE, 2014, pp. 575–589.

[14] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity." in *USENIX Security*, vol. 14, 2015, pp. 28–38.

[15] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2015, pp. 901–913.

[16] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." in *Usenix Security*, vol. 5, 2005.

[17] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.

[18] H. Stuart, "Hunting bugs with coccinelle," *Master's Thesis*, 2008.

[19] S. Rievers, "Finding bugs in open source software using coccinelle," 2009.

[20] D. Kroening, "The cprover user manual. satabs–predicate abstraction with sat. cbmc–bounded model checking," 2008.

[21] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, no. 8. ACM, 2010, pp. 31–40.

[22] J. Caballero, G. Grieco, 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.* ACM, 2012, pp. 133–143.

[23] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification," *Proceedings 2015 Network and Distributed System Security Symposium*, 2015.

[24] Y. Younan, "Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers." in *NDSS*, 2015.

[25] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers." in *USENIX Security Symposium*, 2010, pp. 177–192.

[26] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *Acm sigplan notices*, vol. 41, no. 6. ACM, 2006, pp. 158–168.

[27] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *Proceedings of the 17th ACM conference on Computer and communications security.* ACM, 2010, pp. 573–584.

[28] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

[29] S. Lee, T. Johnson, and E. Raman, "Feedback directed optimization of tcmalloc," in *Proceedings of the workshop on Memory Systems Performance and Correctness.* ACM, 2014, p. 3.

[30] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In proc. of the winter 1992 usenix conference.* Citeseer, 1991.

[31] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[32] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn, "Nozzle: A defense against heap-spraying code injection attacks." in *USENIX Security Symposium*, 2009, pp. 169–186.

[33] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with javascript." *WOOT*, vol. 8, pp. 1–6, 2008.

[34] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.

[35] Y. Brun and M. D. Ernst, "Finding latent code errors via machine learning over program executions," in *ICSE'04, Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, Scotland, May 26–28, 2004, pp. 480–490.

[36] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *Proceedings of the 39th International Conference on Software Engineering.* IEEE Press, 2017, pp. 519–529.

[37] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis," *Static Analysis*, pp. 203–217, 2005.

[38] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *International Static Analysis Symposium.* Springer, 2003, pp. 295–315.

[39] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[40] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction.* ACM, 2016, pp. 265–266.

[41] F. E. Boland Jr and P. E. Black, "The juliet 1.1 c/c++ and java test suite," *Computer (IEEE Computer)*, vol. 45, no. 10, 2012.

[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[43] "Simple support vector machine (svm) example with character recognition." [Online]. Available: https://pythonprogramming.net/support-vector-machine-svm-example-tutorial-scikit-learn-python

[44] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," *2015 IEEE Symposium on Security and Privacy*, 2015.

[45] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar, "Eternal war in memory," *IEEE Security and Privacy*, vol. 12, no. 3, 2014.

[46] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, "Vtpin: practical vtable hijacking protection for binaries," in *Proceedings of the 32nd Annual Conference on Computer Security Applications.* ACM, 2016, pp. 448–459.

[47] M. Payer and T. R. Gross, "Fine-grained user-space security through virtualization," *ACM SIGPLAN Notices*, vol. 46, no. 7, pp. 157–168, 2011.

[48] "Netdata," 2017. [Online]. Available: http://my-netdata.io/