IDENTIFYING AND DOCUMENTING FALSE POSITIVE PATTERNS GENERATED BY STATIC CODE ANALYSIS TOOLS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Zachary P. Reynolds

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2017

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. James H. Hill, Chair

Department of Computer and Information Science

Dr. Rajeev R. Raje

Department of Computer and Information Science

Dr. Mohammad Al Hasan

Department of Computer and Information Science

Approved by:

Dr. Shiaofen Fang

Head of the Graduate Program

To my parents, brothers, and sister.

ACKNOWLEDGMENTS

This research would not have been possible without the following people.

Thank you to Dr. Hill for your direction in leading the project, for motivating me to research solutions on my own, and for encouraging me to tackle challenges with confidence.

Thank you to Dr. Raje for your guidance throughout the project and for relating classroom concepts to our real-world project.

Thank you to Dr. Hasan for your insight while serving on the committee.

Thank you to Abhi for your partnership in the project and for contributing to this research.

Thank you to Nyalia for presenting an abbreviated version of this thesis on my behalf at an international conference.

Thank you to my colleagues Dr. Porter, Dr. Foster, Ugur, Omid, and Parsa for providing feedback on my work and assistance in using the software analysis tools for this project.

Thank you to Brian for your assistance in running the software tools and evaluating their output.

Thank you to Scott and Debbie for setting up the computer environment for my experiments and for providing technical assistance.

Thank you to Dr. Sarkar for your advice on statistics and sampling.

Thank you to Nicole for advising me about numerous logistical details and deadlines throughout the graduate program.

Thank you to the IUPUI Computer Science department for providing the opportunity, funds, and faculty for me to participate on a real-world research project.

Thank you to Dad for your constant support and encouragement; to Mom for always ensuring there is a meal on the table; and to Joshua, Nicholas, and Rachel for sharing my household chores and church responsibilities on countless occasions.

Finally, I thank God for His grace in helping me through this project each day and for motivating me to work to the best of my ability.

This work was sponsored by the Department of Homeland Security under grant #D15PC00169.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Reynolds, Zachary P. M.S., Purdue University, August 2017. Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools. Major Professor: James H. Hill.

Static code analysis tools are known to flag a large number of false positives. A false positive is a warning message generated by a static code analysis tool for a location in the source code that does not have any known problems. This thesis presents our approach and results in identifying and documenting false positives generated by static code analysis tools. The goal of our study was to understand the different kinds of false positives generated so we can (1) automatically determine if a warning message from a static code analysis tool truly indicates an error, and (2) reduce the number of false positives developers must triage. We used two open-source tools and one commercial tool in our study. Our approach led to a hierarchy of 14 core false positive patterns, with some patterns appearing in multiple variations. We implemented checkers to identify the code structures of false positive patterns and to eliminate them from the output of the tools. Preliminary results showed that we were able to reduce the number of warnings by 14.0%-99.9% with a precision of 94.2%-100.0% by applying our false positive filters in different cases.

1 INTRODUCTION

Programmers have relied on two general schools of analysis to enhance software quality: dynamic and static. *Dynamic program analysis* executes programs and evaluates their runtime behavior for testing or profiling [1]. Dynamic analysis exactly determines the presence of flaws since program information is known at runtime, but it suffers from the overhead of program execution. Relying on dynamic analysis alone will expose flaws only late in the development life cycle when they are harder and more costly to fix [2]. On the other hand, *static code analysis* checks programs for errors without actually executing them [3]. Because static analysis does not require a complete, running software program, defects can be found earlier in the software life cycle than with dynamic analysis. Static analysis models programs by considering different control paths and values of variables. Because complex programs have many inputs and execution environments, efficient static analysis cannot account for all possible executions. Instead, an abstraction of the program is used that simplifies analysis while preserving soundness. Static analysis is a sound, conservative approach, meaning that the results hold for all possible executions but may be weaker than the truth. Unlike dynamic analysis, static analysis deals with uncertainty, and therefore results may be imprecise. This thesis focuses on static code analysis in an effort to improve the quality of static code analysis tools.

Developers and testers can use static code analysis to locate flaws in source code that (1) are hard to identify manually and (2) can eventually lead to security vulnerabilities. For example, some flaws detected by static code analysis include buffer overflows, null pointer dereferences, misuse of resource handles, and use of variables before initialization. The MITRE Corporation [4] manages the Common Weakness Enumeration (CWE) project [5], which catalogs weaknesses that can lead to vulnerabilities in software systems. Some weaknesses are abstract (*e.g.*, CWE-118 Improper

Access of Indexable Resource), while others are concrete (*e.g.*, CWE-121 Stack-based Buffer Overflow). The CWE project allows static code analysis tool vendors and users to communicate about the coverage of various analysis tools by using a common language.

There are many static code analysis tools available in the market, both open-source (freely available) and commercial tools [6]. The target programming languages and software weaknesses vary between tools. For example, CAT.NET [7] specializes in detecting security flaws in .NET programs, while FindBugs [8] covers a wider range of CWEs in Java [9]. Some static code analysis tools operate continuously behind-the-scenes within an integrated development environment without user intervention [10], while others may be run as stand-alone tools at the request of users [11].

Irrespective of their focus, a characteristic shared by many static code analysis tools is that they generate large numbers of false positives [12–17] (*i.e.*, the tool generates a warning message that is incorrect). For example, one study evaluated how different static code analysis tools performed on the Juliet test suite [18] from the National Institute of Standards and Technology and discovered a large number of false positives [19]. For two commercial off-the-shelf static code analysis tools used in the study, as many as 59% and 63%, respectively, of the warnings generated were false positives. Other evidence shows that a false positive rate between 30% and 100% is not uncommon even for tools that effectively find errors [16]. In general, we believe the number of false positives generated by these tools is large because (1) static code analysis is *hard* and (2) it is often better for the tool to state that there is a problem and be wrong (*i.e.*, a false positive), than to not state that there is a problem and be wrong (*i.e.*, a false negative).

In either case, there is opportunity to reduce the number of false positives generated by static code analysis tools. This is important because triaging large numbers of false positives is time-consuming for developers and may reduce confidence levels in static code analysis tools. For instance, one study with software developers found false positives to be one of the most significant barriers to using static code analysis

tools [11]. Researchers found that a large list of warnings with false positives discouraged developers from using the tools in the first place. Instead, developers need to focus on generated warnings that are true warnings and address them properly. In order to reduce the number of false positives, however, we must first understand the different kinds of false positives generated by static code analysis tools. This approach is similar to the way in which CWEs characterize different kinds of vulnerabilities in source code. CWEs organize repeated instances of software weaknesses, and we aim to use the same approach to organize false positive warnings into patterns.

We hypothesize the following:

1. *Static code analysis tool false positives occur as recurring patterns.*

2. *These patterns can be harnessed to reduce the number of false positives in static code analysis tools.*

Our approach is as follows: We ran a set of static code analysis tools over a code base to identify false positives. Next, we reduced the test cases containing false positives to a minimized form, giving the core structure causing the false positive. Then we cataloged the false positives to learn patterns. Finally, we validated our work with static code analysis tool developers by having them provide feedback on our false positive patterns.

With this understanding, the contributions of this thesis are as follows:

- We show that static code analysis tools flag few recurring false positive patterns when applied to a standardized test suite, such as the Juliet test suite.

- We standardize how to define false positive patterns using a set of descriptive attributes. These attributes include the following statistics: the false warning message flagged by the tool; a measure of how often the pattern occurs in tested source code; a minimized source code snippet showing the essence of the false positive; and an informal description of the pattern.

- We create a hierarchical catalog of false positive patterns to better understand its structure and the variation between similar and different false positive patterns.
- We show the practicality of using a standardized test suite to identify false positive patterns.
- We design a filter for identifying false positive patterns and for eliminating them from a static code analysis tool's output.

We performed our study in the context of both open-source and commercial static code analysis tools available from both academia and industry. The results of our study produced a catalog of 14 different false positive patterns, some of which we validated with static code analysis tool developers.

This thesis is organized as follows: Chapter 2 discusses existing approaches from the literature for eliminating false positives in static code analysis tools; Chapter 3 describes our approach for identifying and documenting false positive patterns; Chapter 4 presents our catalog of false positive patterns and other results from documenting false positive patterns; Chapter 5 presents our preliminary work on filtering out false positives from a tool's output based on the information in our catalog; and Chapter 6 concludes the thesis. Finally, Appendix A provides an abbreviated version of our false positive catalog, and Appendix B describes our implementation details for filtering out instances of one false positive pattern.

2 RELATED WORK

This chapter discusses other approaches in the literature for reducing the number of false positive reports from static code analysis tools.

2.1 Machine Learning

A number of studies have applied machine learning techniques on characteristics describing the warning and its context obtained via static code analysis tools. Yuksel and Sozer [20] developed a binary classifier to distinguish between true and false warnings based on 10 attributes, including the warning severity, type of warning, number of warnings in the file, and length of time the warning has persisted through consecutive runs of the static code analysis tool. While the authors concluded their approach to be viable, their classifier still depended on the developer's initial perception of the error—whether it was an actual error or could be ignored. Similarly, Heckman and Williams [21] applied machine learning techniques to predict false warnings based on a number of software and revision control metrics. Both the ideal machine learning technique and the set of important metrics for classification needed to be tuned to the specific software program being analyzed. Unlike these prevalent approaches, this thesis uses the structure of the source code to identify false positives.

Tripp et al. [17] also addressed the problem of false positives with a variety of machine learning techniques. Their model, which was named Eletheia, required the user to manually classify a subset of the warnings. Eletheia then automatically constructed a statistical filter and ran the filter over the remaining warnings, removing warnings classified as false positives. Unlike other automated approaches, Eletheia allowed the user to customize the filter to prioritize either removal of false positives or preservation of true positives. The learning features used by Eletheia included

a limited number of structural aspects, such as function names of data sources and sinks, but this thesis focuses solely on structure to classify false positives. In addition, the creation of a false positive catalog in this thesis enables identification of high-level constructs that tend to generate false positives.

Koc et al. [22] trained both a Bayesian classifier and a long short-term memories (LSTM) neural network to remove false positives in Java code. The authors trained their models on bytecode instructions, which are simplified and easier to analyze with machine learning techniques as compared to source code. They showed that it was possible to infer structures tending to generate false positives, but these were necessarily micro-level structures since they were based on individual bytecode instructions. This thesis seeks to identify macro-level structures based on the source code.

2.2 Formal Methods

Other methods of eliminating false positives examine the software program itself for each individual warning using formal methods. Arzt et al. [23] created a symbolic execution program to remove dataflow false positives from a static code analysis tool output for Android applications. Their filtering mechanism, known as TASMAN, scanned conditionals along a reported warning path for logical contradictions. Warnings with paths that were provably impossible were eliminated as false positives. The authors observed that many conditionals were not local and could therefore not be thoroughly analyzed by a trivial analysis, producing these false positives. This false positive elimination approach gave perfect precision (no actual errors were eliminated) on both artificial benchmarks and real-world applications and eliminated 80% of the false positives for one benchmark. Unlike the approach in this thesis, TASMAN is fully automated, but it suffers from considerable processing time, requiring over 5 seconds on average to evaluate each warning. Their approach also has limited applicability because it can be applied only when values of variables can be statically

determined from the source code. This thesis addresses this challenge by allowing any false positive to be eliminated if its structure has been identified.

Muske et al. [24] augmented static analysis with model checking to improve precision. In their work, a model checker generated assertions associated with each static code analysis tool warning. Warnings associated with assertions that could be verified were eliminated as false positives. As in [23], the verification process was slow, but the authors introduced a set of optimizations to avoid verifying assertions that would not eliminate additional false positives. These optimizations reduced processing time by 60%. Unlike formal methods, this thesis relies on the hypothesis that patterns of false positives exist that can be harnessed to quickly eliminate similar false positives in other contexts.

2.3 Pattern Identification

Finally, some work has been done to classify warnings according to patterns. Ayewah et al. [15] manually classified warnings from FindBugs [8] on several open-source projects into three classes: false positives, trivial bugs, and serious bugs. Trivial bugs indicated actual defects but had little to no impact on the application's functionality. The authors reported 6 categories of trivial bugs and found that many of these bugs were intentional and/or the result of poor programming practices. These 6 categories were high-level and did not describe the code structure, unlike this thesis. For example, one category of trivial bugs from the authors' study was *testing code*, where FindBugs flagged an unusual situation in a test case, but the test case was specifically designed to test for the unusual situation.[1] The authors therefore classified this category as a trivial bug. This thesis builds on their work by cataloging false warnings according to code structure with the intent of automatically identifying false positive patterns in other contexts.

[1] An example given in the paper was to check that `.equals(null)` returns false to ensure that the `equals()` method could handle a null argument.

Beller et al. [25] recently proposed a classification scheme for both software defects and warnings from static code analysis tools. Their *General Defect Classification* (GDC) scheme was designed to map warnings from different static code analysis tools to the same classification hierarchy. However, the scheme's classification categories are too broad and at a high-level to classify false positives with respect to structure. For example, while the scheme may distinguish between resource and logic warnings—two categories of the GDC scheme—these categories do not account for the different code structures that may generate resource or logic warnings. This thesis seeks a more detailed classification hierarchy that can capture the structure of the source code. This thesis also focuses on cataloging patterns of false warnings, instead of cataloging all warnings from static code analysis tools.

3 APPROACH

This chapter describes our approach to identifying false positive patterns. Figure 3.1 provides an overview of our approach. As shown in this figure, our approach involved the following steps:

1. We selected the set of static code analysis tools to use;
2. We selected the codebase over which to run the static code analysis tools;
3. We ran the static code analysis tools over the codebase to identify false positives;
4. We reduced the source code files generating false positives to a minimized form;
5. We cataloged the false positives to learn false positive patterns; and
6. We provided feedback to the static code analysis tool developers to validate our work.

Each step in our process is discussed in more detail in the following sections.

3.1 Selecting the Static Code Analysis Toolset

The first step was to select the set of static code analysis tools to use in our experiment. We selected three static code analysis tools, as listed in Table 3.1. We have removed the tool names in accordance with our agreement with the tool vendors. We included two open-source tools because they are freely available and provide a base case to compare against other tools. We also included a state-of-the-art commercial tool because commercial tools are considered to be more reliable than open-source tools [26].

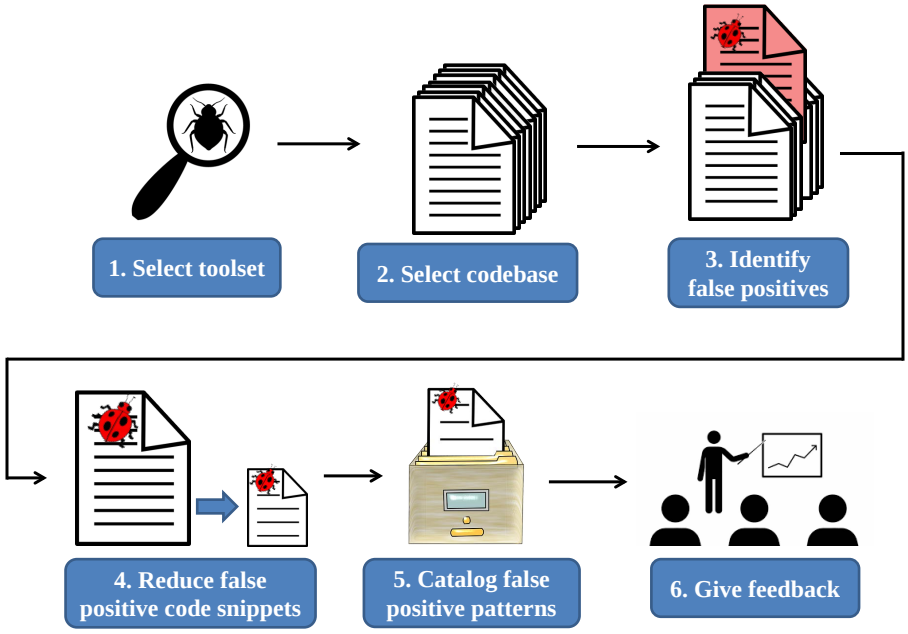

Figure 3.1.: General approach for identifying false positives generated by static code analysis tools.

Table 3.1.: Static code analysis tools selected for our experiment.

Static code analysis tool	Tool type
Tool A	Open source
Tool B	Open source
Tool C	Commercial

3.2 Selecting the Codebase for Analysis

The next step in our process was to select the codebase for our study. We selected the C/C++ Juliet test suite version 1.2 [18] from the National Security Agency (NSA) Center for Assured Software. We selected the Juliet test suite because it was designed specifically for evaluating how static code analysis tools perform against known weaknesses in source code that can lead to security vulnerabilities. While the Juliet test

suite also contains a Java version, we limited our scope to the C/C++ version. The C/C++ languages are generally considered less secure and more vulnerable to certain flaws, such as memory corruption, that are prevented in Java by its type system [27]. We therefore believed C/C++ would provide greater insight into our experiment.

The Juliet test suite contains 61,387 individual test cases grouped according to 118 different weaknesses as identified by the MITRE's Common Weakness Enumeration (CWE) [28]. Each CWE is numbered arbitrarily and assigned a name that reflects the concern being documented. For example, CWE-457 is named "Use of Uninitialized Variable," which implies the weakness documents code that uses a variable that has not been initialized. Likewise, CWE-415 is named "Double Free," which implies the weakness documents code that deallocates the same memory location twice. Each test case expresses a flaw of a particular CWE, which is said to be the weakness that the test case *targets*.

Juliet test cases are further organized by *flow variants*, which are different control and data flows for expressing the same flaw. Similar to CWEs, flow variants are numbered. For example, the first flow variant—referred to as *baseline*—captures the simplest form of the flaw without any additional complexity. Likewise, flow variant 9 controls the program flow with a conditional that evaluates a constant global variable, while flow variant 72 passes data from one function to another via a vector.

One advantage of using Juliet for our study is that each test case is annotated with the following information:

- **Flaw.** A `FLAW` annotation denotes the location of the error of the target weakness.
- **Potential flaw.** For target weakness that involve both sources (*i.e.*, where the flaw originates) and sinks (*i.e.*, where the error actually arises), `POTENTIAL FLAW` annotations are used in both locations. For example, for a weakness involving tainted data, the source may read untrusted data from the user, while the sink

may perform a security-critical operation with that data, such as a database write [17].

- **Fix.** When possible, the Juliet test suite includes a non-flawed version of a test case that performs the same functionality as the flawed version, but does not contain the target weakness. In this case, a `FIX` annotation denotes a change to the source code to remove the flaw.

- **Incidental.** An `INCIDENTAL` annotation marks an unavoidable flaw whose type is *different* from the target weakness.

The annotations in Juliet allow us to quickly identify true positives, false positives, and false negatives generated by a static code analysis tool. For example, Listing 3.1 shows a snippet of a Juliet test case with a target error of CWE-369 Divide By Zero, slightly simplified for readability. This code snippet contains a divide-by-zero flaw that can occur for some user input.[1] The first `POTENTIAL FLAW` annotation on line 9 marks the source of the flaw, while the second annotation on line 19 marks the sink of the flaw.

Listing 3.2 on page 14 is a non-flawed version of the test case given in Listing 3.1. An appropriate fix to this program could be a change to either the source or sink of the flaw. Listing 3.2 changes the source by initializing the `data` variable to a known non-zero value and thus avoids the divide-by-zero error. The `FIX` annotation on line 12 denotes this change. The sink was not changed, so the `POTENTIAL FLAW` annotation remains on line 16. Note then that a `FIX` annotation in the same test case as a `POTENTIAL FLAW` annotation indicates a non-flawed version. The `INCIDENTAL` annotation on line 8 of Listing 3.2 marks a legitimate dead code error not of the target error type. Because the `0` in the condition on line 7 will evaluate to false, the `printLine()` statement will never execute. Therefore, the statement is dead code.

[1] Any user input that evaluates to 0 will cause a divide-by-zero error.

```
1  // CWE369_Divide_by_Zero__int_fgets_divide_02.c
2  ...
3  void CWE369_Divide_by_Zero__int_fgets_divide_02_bad() {
4    int data;
5    /* Initialize data */
6    data = -1;
7    if(1) {
8      char inputBuffer[CHAR_ARRAY_SIZE] = "";
9      /* POTENTIAL FLAW: Read data from the console using fgets() */
10     if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL) {
11       /* Convert to int */
12       data = atoi(inputBuffer);
13     }
14     else {
15       printLine("fgets() failed.");
16     }
17   }
18   if(1) {
19     /* POTENTIAL FLAW: Possibly divide by zero */
20     printIntLine(100 / data);
21   }
22 }
23 ...
```

Listing 3.1: Snippet of a Juliet test case showing flaw annotations.

Another advantage of the Juliet test suite is that the flaws are simplified versions of problems that occur in more complex source code. This makes it easier to understand the source code and corresponding software weaknesses.

3.3 Identifying the Generated False Positives

As the next step, we ran the static code analysis tools over the Juliet test suite to identify false positives. We define a *false positive* as a warning message generated by a static code analysis tool for a location in the source code that does not have any known problems. Other studies have defined false positives more broadly. Heckman and Williams [21] considered warnings that indicate actual defects but which pose no

```
1  // CWE369_Divide_by_Zero__int_fgets_divide_02.c
2  ...
3  static void goodG2B1() {
4    int data;
5    /* Initialize data */
6    data = -1;
7    if(0) {
8      /* INCIDENTAL: CWE 561 Dead Code */
9      printLine("Benign, fixed string");
10   }
11   else {
12     /* FIX: Use a value not equal to zero */
13     data = 7;
14   }
15   if(1) {
16     /* POTENTIAL FLAW: Possibly divide by zero */
17     printIntLine(100 / data);
18   }
19 }
20 ...
```

Listing 3.2: Snippet of a Juliet test case showing a fix annotation.

significant threat to a program's functionality as false positives. For example, a warning could indicate a potential maintainability issue that does not affect functionality, and addressing the warning could arguably introduce other more serious defects. In this case, the warning would be deemed *unactionable* since the developer should not take action to fix it. To provide a more objective analysis, however, we focused on warnings that did not indicate actual defects.

To better understand the problem, Listing 3.3 shows a simplified code snippet from the Juliet test suite that causes Tool A to generate a false positive. The code snippet initializes a `char` array to the empty string. Data is then read from `stdin` using `fgets()` and stored in the array variable. A maximum of 9 characters will be read in addition to the null character. Because the program writes to the beginning of the array and the array has space for 10 characters, all memory accesses will be valid. It is also possible that fewer than 10 characters will be written based on the

input, but no more than 10. Therefore, no write will occur outside the bounds of the buffer. Unfortunately, Tool A warns that the buffer is accessed out of bounds during the write operation.[2] (We have added the "Warning" annotation in the source code to indicate this warning message, as we have done with other code listings throughout this thesis.) We regard this warning as a false positive.

```
1  // CWE121_Stack_Based_Buffer_Overflow__CWE129_fgets_01.c
2  // Pattern: buffer-underflow-usage
3  #include <stdio.h>
4  #define BUFFER_SIZE 10
5
6  int main(void) {
7    char buffer[BUFFER_SIZE] = "";
8    // Warning: Buffer is accessed out of bounds.
9    fgets(buffer, BUFFER_SIZE, stdin);
10   return 0;
11 }
```

Listing 3.3: False positive example from a simplified Juliet test case.

This example is a trivial program. For more complicated programs, we must consider multiple input scenarios to determine whether a warning can occur or not. For example, consider a buffer overflow warning flagged by a static code analysis tool. Suppose only an unusual input to the program will actually cause a buffer overflow at the warned location, while all other inputs cause the program to execute safely. Buffer overflows are serious flaws that can compromise data integrity and confidentiality [29]. Therefore, even though the warning may not be realized for some inputs, we regard such a warning to be a true positive.

There are also cases where a static code analysis tool flags an error that cannot occur for the given calling context but may occur for a different context. For example, consider Listing 3.4, which is another code snippet from the Juliet test suite. Tool C warns that a buffer overrun occurs while reading an element of the list on line 7.

[2]Note that adding a check for the return value of `fgets()` still causes the static code analysis tool to display a warning message.

However, this program will not cause a buffer overrun because an item is pushed onto the list before being accessed in the `good()` function, and only the `main()` function invokes `good()`. If the `good()` function, however, is executed in a different context (*e.g.*, the `push_back()` call is omitted on line 13), the warning may occur. Therefore, the buffer overrun will not occur with the program implemented "as-is," but the warning may be legitimate in another context. This false positive is introduced due to *context insensitivity* [17], an approximation made by some static code analysis tools to achieve scalability at the expense of loss of precision. We still regard an example like this to be a false positive for the given context.

```
1  // Juliet CWE476_NULL_Pointer_Dereference__char_73b.cpp
2  // Pattern: list-overrun
3  #include <list>
4
5  void good(std::list<char *> dataList) {
6    // Warning: Buffer overrun.
7    char * data = dataList.back();
8  }
9
10 int main(void) {
11   char * str = "test string";
12   std::list<char *> dataList;
13   dataList.push_back(str);
14   good(dataList);
15   return 0;
16 }
```

Listing 3.4: False positive example from the Juliet test suite under a particular execution context.

Unfortunately, there is no easy (or automated) method for identifying in general whether a generated warning message is indeed a false positive or not. Fortunately, we are using the Juliet test suite, which makes it possible to automatically identify a false positive. For example, a warning message is a false positive in the context of the Juliet test suite when there are no annotated flaws on the same line flagged by

the warning message. In addition, other warnings are false positives if they are of the same type targeted by the test case but are flagged in a fixed version of the test case.

We therefore leveraged prior research from our group on evaluating static code analysis tools to quickly locate false positives generated by each static code analysis tool used in our study. More specifically, we integrated each static code analysis tool into the Static Code Analysis Tool Evaluator (SCATE) [19], a framework for evaluating the quality of static code analysis tools. SCATE uses rules similar to the foregoing to automatically classify static code analysis tool warnings in the Juliet test suite. Next, we executed the static code analysis tools against the Juliet test suite and filtered out warnings classified as false positives by SCATE. After that, we verified that the warnings were indeed false positives. Note that because of the sheer number of warnings classified as false by SCATE, we sampled SCATE's output. For our verification process, we inspected the source code and executed the test cases with dynamic profiling tools, such as Valgrind [30], which can indicate the presence of runtime errors (*e.g.*, memory leaks, out-of-bound reads or writes, *etc.*).

If our careful inspection and dynamic analysis indicated that the warning was a false positive, then we concluded the warning to be a false positive for the given context. We then studied the program's input to determine the context of the false positive. A warning may be a false positive for a given execution but a true positive in other contexts. For example, the warning generated by a static code analysis tool in Listing 3.4 on the previous page is false for the given context (with an element pushed to the list) but true if an empty list is passed.

To illustrate this verification process, Listing 3.5 shows a portion of a Juliet test case with a false positive. Lines 6 and 7 declare a pointer and a reference to a pointer, respectively. Next, the value `Good` is assigned to the pointer location. Finally, on lines 12-14, the value is read through the reference-to-pointer and printed. One static code analysis tool reports a warning on line 9 of Listing 3.5 that a stored value is

never read. Executing the program shows that the value “G” is actually read and printed in its hexadecimal form 47.[3] This warning is therefore a false positive.

```
1  #include "std_testcase.h"
2  #include <wchar.h>
3  ...
4  #ifndef OMITGOOD
5  static void goodG2B() {
6    char * data;
7    char * &dataRef = data;
8    /* FIX: Initialize data */
9    // Warning: Value stored to 'data' is never read.
10   data = "Good";
11   {
12     char * data = dataRef;
13     /* POTENTIAL FLAW: Attempt to use data, which may be NULL */
14     printHexCharLine(data[0]);
15   }
16 }
17 ...
18 void good() {
19   goodG2B();
20   goodB2G();
21 }
22 #endif /* OMITGOOD */
23 ...
24 int main(int argc, char * argv[]) {
25   ...
26   good();
27   ...
28   return 0;
29 }
```

Listing 3.5: Example false positive warning in a Juliet test case.

[3]Note that printing the value as character data instead of as hexadecimal data does not change the static code analysis tool warning.

3.4 Reducing the Source Code

After finding a false positive message, we manually reduced the source code that generated the false positive. By *reduced*, we mean we removed all elements of the source code not related to the false positive until the removal of the next element resulted in the false positive message disappearing. We reduced the source code because it allowed us to better identify the cause of the false positive since only the essence of the problem remained. We refer to the reduced source code, along with a set of descriptive attributes defined in Section 3.5, as a *false positive pattern*. For example, Listing 3.6 shows the reduced code for Listing 3.5. It should also be noted that this reduction accomplishes more than is first apparent, because the original test case in Listing 3.5 was 118 lines long, and we reduced it to only 10 lines; we showed only the most relevant code in Listing 3.5 due to space limitations. Here are key observations about our reduction:

- The false positive was still present in the reduced code. This is a critical point and is true for all reduced false positives. In other words, running the static code analysis tool on the reduced code still gave the same warning, and the warning was false. There are two other possibilities after reduction: (1) the analysis tool flagged the same warning message, but now the warning was true, or (2) the analysis tool no longer flagged the warning. Either case indicated we reduced the code too far, requiring us to backtrack to the previous state of the program.

- Unnecessary functions were removed, and relevant functions were renamed to meaningful names. In the original code of Listing 3.5, the program flow transitioned from `main()` to `good()` to `goodG2B()`. As the reduced code shows, `main()` and `good()` were not relevant to the false positive, so they were removed. We discovered unnecessary functions by trial and error.

- Reduction within relevant functions was performed. The code on lines 12-14 in the `goodG2B()` function of Listing 3.5 was moved out of the nested brackets, and the data was read directly from the `dataRef` reference.

- Dependence on domain-specific code libraries was eliminated. The header `std_testcase.h` (see line 1 of Listing 3.5) defines reusable types and functions that test cases depend on. We removed this dependency and substituted these types and functions for standard C/C++ ones. For example, the `printHexCharLine()` call on line 14 was replaced with a `printf()` call. The reduced code was more self-contained and allowed us to focus on the code structures relevant to the false positive.

```
1  #include <stdio.h>
2
3  int main(void) {
4    char * data;
5    char * &dataRef = data;
6    // Warning: Value stored to 'data' is never read.
7    data = "Good";
8    printf("%c\n", dataRef[0]);
9    return 0;
10 }
```

Listing 3.6: Reduced version of Listing 3.5.

Listing 3.7 on the following page and Listing 3.8 on page 22 show the original code and reduced code, respectively, of another false positive. Beginning on line 19 of the original code, the `goodG2B2()` function allocates a `char` array on the stack and defines a `char` pointer `data` to point to the array. The `data` pointer is passed to the `goodG2B2Source()` function, where the data is initialized and the pointer is returned. One static code analysis tool generates a warning that the `goodG2B2Source()` function returns a pointer to a local variable. The returned `data` pointer, however, was itself passed as a parameter to the `goodG2B2Source()` function. The pointer there-

```
1  #include "std_testcase.h"
2  #include <wchar.h>
3  #ifndef OMITGOOD
4  /* The static variables below are used to drive
5  control flow in the source functions. */
6  static int goodG2B2Static = 0;
7  ...
8  /* goodG2B2() - use goodsource and badsink by reversing
9  the blocks in the if in the source function */
10 static char * goodG2B2Source(char * data) {
11   if(goodG2B2Static) {
12     /* FIX: Properly initialize data */
13     data[0] = '\0'; /* null terminate */
14   }
15   // Warning: Pointer to local array variable returned.
16   return data;
17 }
18 static void goodG2B2() {
19   char * data;
20   char dataBuffer[100];
21   data = dataBuffer;
22   goodG2B2Static = 1; /* true */
23   data = goodG2B2Source(data);
24   {
25     char source[100];
26     memset(source, 'C', 100-1); /* fill with 'C's */
27     source[100-1] = '\0'; /* null terminate */
28     /* POTENTIAL FLAW: If data is not initialized
29     properly, strcat() may not function correctly */
30     strcat(data, source);
31     printLine(data);
32   }
33 }
34 void CWE665_Improper_Initialization__char_cat_21_good() {
35   goodG2B1();
36   goodG2B2();
37 }
38 #endif /* OMITGOOD */
39 ...
40 int main(int argc, char * argv[]) {
41   ...
42   CWE665_Improper_Initialization__char_cat_21_good();
43   ...
44   return 0;
45 }
```

Listing 3.7: Original code of *Return Local Param* false positive pattern.

```
1  // CWE665_Improper_Initialization___char_cat_21.c
2  // Pattern: return-local-param
3  char * helper(char * data) {
4    // Warning: Pointer to local array variable returned.
5    return data;
6  }
7
8  int main(void) {
9    char * data;
10   char dataBuffer[10];
11   data = dataBuffer;
12   data = helper(data);
13   return 0;
14 }
```

Listing 3.8: Reduced code of *Return Local Param* false positive pattern.

fore still has meaning to the caller within `goodG2B2()`, making this warning message a false positive message.

The reduction methods used in the first example of this section were also applied to this example. We removed unneeded functions, reduced within the relevant functions `goodG2B2()` and `goodG2B2Source()`, renamed these functions to meaningful names, and removed dependence on Juliet test case support files. The initialization performed inside `goodG2B2Source()` was also irrelevant to the false positive because the static code analysis tool still generated the warning if the initialization was removed. The reduced code is therefore more succinct than the original code. It also clearly shows the structure that causes the false positive—returning the `data` pointer parameter from the `helper()` function. As in the previous example, moving from the original code (156 lines) to the reduced code (12 lines without the added header comments) demonstrates significant reduction and isolates the source of the false positive.

3.5 Cataloging Patterns

After reducing the source code that produced a false positive message by a static code analysis tool, we cataloged the reduced code along with a few descriptive attributes. Recall that the reduced code with the attributes define the false positive pattern. The goal of cataloging the false positive pattern was to identify recurring source code structures in the codebase that cause static code analysis tools to generate false positives. This approach is similar to documenting software design patterns [31] or cataloging software weaknesses [28]. In all these scenarios, the goal is to capture similarities of many instances so the knowledge can be documented and applied elsewhere.

Each documented false positive pattern in our catalog has the following attributes:

- **Name**. The name is a short phrase describing the code structure of the false positive pattern.
- **False positive warning**. The false positive warning is the warning message from the static code analysis tool, or tools. We say *tools* because some false positive patterns were identified by multiple tools. If multiple tools flagged the pattern, we recorded an abstract warning message. For example, if Tool A gave an "uninitialized variable: username" false warning while Tool B gave a "function call argument is an uninitialized value" false warning for the same use of a variable, we abstracted the warning to simply "uninitialized variable."
- **Description**. The description is an informal, high-level explanation of the code structure causing the false positive.
- **Tools**. This attribute records the list of static code analysis tool(s) which reported the false positive pattern.
- **Frequency**. Frequency is a measure of how often the false positive appeared in the codebase. As one measure, we recorded the number of unique occurrences

of the pattern as a basic frequency attribute.[4] In addition, certain software domains may benefit from other attributes for indicating a pattern's prevalence. For example, the Juliet test suite organizes test cases according to two independent attributes that can provide more detailed frequency information: CWEs and flow variants (both described in Section 3.2). We recorded both CWEs and flow variants in which the false positive was present to determine if a false positive pattern was limited to a specific context or widespread. This is another benefit of using a standardized, well-organized test suite such as Juliet.

- **Code snippet**. We included a reduced C/C++ code snippet of the false positive using the reduction method discussed in Section 3.4. We commented each code snippet with the name of the test case in Juliet from which it was derived. We also included the false warning flagged by the analysis tool immediately before the line flagged by the tool. If multiple tools flagged the same warning, we listed each tool, along with the exact text of the warning messages.

- **Changes**. Changes are minor modifications to the reduced code snippet, along with the static code analysis tool result. We made these changes intuitively with the goal of having the tool no longer flag the warning. Some changes caused the false warning to disappear, others did not, but each change still provided important insight. By comparing the reduced code with the changes, we can better understand the code structures causing the false positive. We stored both the changed source code snippet(s) and the results from the static code analysis tools, in addition to the unchanged (reduced) code snippet.

For example, we cataloged the *Buffer Underflow Usage* false positive pattern as follows:

- **False positive warning**: Buffer is accessed out of bounds.

[4]Some patterns occurred often, so we relied on sampling to obtain a frequency approximation. See Section 4.4 for more information.

- **Description**: A value is stored to a buffer after the buffer is initialized. Let L be the length of the value that is stored. The buffer must have been initialized to a value of length less than L-1.
- **Tools**: Tool A.[5]
- **Frequency**: 3,685 occurrences. Occurs in the following CWEs: 127, 134, 190, 194, 195, 197, 226, 367, 369, 400, 517, 617. Not specific to any particular flow variant.
- **Code snippet**: Listing 3.9 gives the reduced code snippet for this pattern. This reduced code is the same as in Listing 3.3 on page 15, but we annotated the code with the names of the static code analysis tools that flagged the false positive. In this case, only Tool A generated a false positive. We also annotated on the first two lines (1) the filename in the Juliet test suite from which we extracted the pattern and (2) the name we designed for the false positive pattern, respectively.
- **Changes**: Increasing the length of the buffer's initial value to L-1 causes the tool to no longer flag the false positive.

We performed the process in this chapter iteratively for each false positive we inspected, producing a total of 27 false positive patterns. We stored the catalog of false positive patterns in a GitHub repository.[6] If a newly reduced false positive structure matched an existing pattern in the catalog, then we did not create a new pattern. Instead, we updated the existing pattern's frequency attribute. If a reduced structure was *similar* to an existing pattern but had no match, then we created a generalized description of the false positive with the concrete patterns as variants (or *child patterns*) of the general pattern. This approach is similar to object-oriented design: the general pattern captured the similarities of its child patterns, while the

[5]The name of the tool is removed to achieve anonymity.

[6]The GitHub repository can be found at the following location: `https://github.com/SEDS/mangrove-catalog/wiki`. An abbreviated version of the catalog is provided in Appendix A of this thesis.

```
1  // CWE121_Stack_Based_Buffer_Overflow___CWE129_fgets_01.c
2  // Pattern: buffer-underflow-usage
3  #include <stdio.h>
4  #define BUFFER_SIZE 10
5
6  int main(void) {
7    char buffer[BUFFER_SIZE] = "";
8    // Tool A FP: Buffer is accessed out of bounds.
9    // Tool B FP: None
10   // Tool C FP: None
11   fgets(buffer, BUFFER_SIZE, stdin);
12   return 0;
13 }
```

Listing 3.9: Reduced code snippet for the *Buffer Underflow Usage* false positive pattern.

child patterns specified concrete details that distinguished them from other child patterns. As a result, we produced a hierarchy of false positive patterns, as illustrated in Section 4.1.

3.6 Giving Feedback to Static Code Analysis Tool Developers

The last step in our approach was to validate the identified false positive patterns by submitting our findings to static code analysis tool developers. We then requested the developers to review the submitted artifacts and confirm that each false positive was indeed a false positive. Section 4.3 discusses our results from this validation.

To summarize this chapter, we selected a toolset of two open-source static code analysis tools and one commercial tool. We selected the Juliet test suite for our codebase because it is designed for testing static code analysis tools, is annotated with the locations of software weaknesses, and enables automated classification of warnings from tools. We used our automated classifier on the output of the tools to filter out false positives, which we manually verified to be false. Then we manually reduced the source code snippets producing false positives to minimal forms in order

to identify core structures leading to false positives. Next, we cataloged the false positives with their associated code snippets, false warning messages, descriptions, tool names, frequencies, and other information to produce false positive patterns. Finally, we validated our work by submitting false positive patterns from our catalog to the tool developer industry for feedback.

4 RESULTS AND DISCUSSION

This section discusses our results from identifying and documenting false positive patterns generated by static code analysis tools.

4.1 False Positive Hierarchy

Figure 4.1 shows a hierarchical view of the false positive patterns that we have identified and documented.[1] The patterns are organized as nodes and are pictured with their names from our catalog. As captured in Figure 4.1, some patterns are variations of one another—hence the multiple layers. At the first column, there are 14 distinct false positive patterns that have no common code structure; we refer to these as *pattern families*. The false positive patterns in the right-hand columns are variations of these core patterns—producing a total of 27 concrete false positive patterns (leaf nodes in Figure 4.1).

Nodes are colored according to which static code analysis tools flagged the false positive patterns. Gray nodes are abstract false positive patterns; that is, they abstract out a common structure from multiple false positive patterns. We cataloged code snippets only for colored (leaf) patterns. For example, the *Operation Through Alias* abstract pattern performs an operation on a resource through an alias, with the exact operation determined by the concrete pattern: one child concrete pattern *Ref Ptr Read* reads a character array through a reference to a pointer, while another child concrete pattern *Subscope Leak* deallocates a memory block in a subscope (*i.e.*, in a different scope that is still visible to the scope that performed the allocation).

Sometimes concrete patterns of the same abstract pattern resulted in different static code analysis tool warnings, as with the *Operation Through Alias* pattern. For

[1] An abbreviated form of the false positive catalog is included in Appendix A.

Figure 4.1.: Hierarchy of identified false positive patterns when executing static code analysis tools against the Juliet test suite. The 14 core false positive patterns or *pattern families* are numbered in the first column.

Ref Ptr Read, Tool B warned that a stored value was never read, while for *Subscope Leak*, Tool A warned that a memory leak occurred. In this case, we did not include a warning at the abstract level since it differed among the concrete patterns.

In other cases, as with *Buffer Store*, the warning was the same for all concrete patterns of the abstract false positive pattern. The abstract *Buffer Store* pattern read data from a source and wrote it safely to a buffer. However, Tool C warned that the buffer write was tainted and could result in a buffer overflow. Each of the three child concrete patterns of the *Buffer Store* pattern used a different source for the data: either a file, a socket, or an environment variable. For each of these sources, the input data was properly trimmed according to the length of the buffer, so the warning was a false positive.

Some false positive patterns were prevalent across multiple tools, while others were specific to a single tool. Most of the concrete child patterns of the *Predictable Conditional* abstract pattern, for example, occurred in both Tools A and B. On the other hand, most of the other patterns were flagged only by a single tool, such as *Array Input Conditional*, *Buffer Underflow Usage*, and *Null Array Access*, to name a few.

In addition to the structural relationships depicted in Figure 4.1, we noticed an orthogonal manner in which patterns aligned to layers. We observed two levels of pattern abstractions in our catalog: lower-level implementation concerns and higher-level design concerns. Many of the child patterns of the *Predictable Conditional* pattern, for example, are at the design level because they involved global variables that analysis tools could not correctly reason about. This situation advised a design refactor, such as replacing the use of global variables with the Singleton design pattern [32]. In addition, the *Array Input Conditional* pattern included an important edge case that confused the analysis tool, which incorrectly indicated the edge case was unnecessary. Because of the suggested design change, we regard this pattern to be at the design level as well. On the other hand, the patterns *List Overrun* and *Sscanf Uninit Var* are examples of implementation-level patterns. These two false

positives could be addressed by static code analysis tool users by performing a simple boundary check or return value check, respectively. The amount of effort needed to address these situations and other implementation-level concerns is minimal. Indeed, from the tool developer's perspective, the aforementioned user action may resolve the false positive issue completely. Thus, compared to design-level patterns, we regard implementation-level patterns as having low severity. Understanding the significance of these abstractions and their correlation with severity will continue to be explored in a future research direction.

4.2 False Positive Pattern Examples

In this section, we describe and give code examples of a few false positive patterns from our catalog.

4.2.1 *Conditional Mem Leak* False Positive Pattern

As we tested different static code analysis tools, it became clear that global variables were a significant source of false positive warnings, particularly with the two open source tools. The commercial tool was able to more accurately analyze control flow that involved global variables and therefore avoid generating false positives in these cases. The following six patterns in our catalog are associated with global variables—specifically global variables used in conditions:

- Conditional Mem Leak
- Conditional Mem Leak External Var
- Conditional Null Ptr
- Conditional Uninit Var
- Conditional Uninit Var External Var

- Intermediate Function Call

In Listing 4.1, we provide an example of a global variable-related false positive pattern by way of the *Conditional Mem Leak* pattern. This listing shows the reduced code (the original program was 150 lines long), which we have annotated with false warnings from the static code analysis tools. The code snippet first allocates memory for a `char` array. Next, the condition tests the value of a global variable, and the memory is freed if the global variable evaluates to `true`. Because the global variable is initialized to `1` and never changed, static code analysis tools should be able to analyze that the memory will be freed and no memory leak will occur. Two static code analysis tools, however, flagged a memory leak false positive for this example, as shown by our annotations on lines 11 and 12.

```
1  // CWE761_Free_Pointer_Not_at_Start_of_Buffer__wchar_t_fixed_string_05.c
2  // Pattern: conditional-mem-leak
3  #include <stdlib.h>
4  static int staticTrue = 1;
5
6  int main(void) {
7    char * data = (char *)malloc(10*sizeof(char));
8    if(staticTrue) {
9      free(data);
10   }
11   // Tool A FP: Memory leak: data
12   // Tool B FP: Potential leak of memory pointed to by 'data'
13   // Tool C FP: None
14 }
```

Listing 4.1: Reduced code of the *Conditional Mem Leak* false positive pattern.

4.2.2 *File Close Virtual Method* False Positive Pattern

Some false positive patterns, such as *Alloc Using Property* and *File Close Virtual Method*, involve user-defined classes. Listings 4.2, 4.3, and 4.4 show the reduced code for the latter pattern. Listing 4.2 defines a `FileCloserBase` base class

with a virtual `closeFile()` method that has an empty implementation. The class `FileCloserSubclass` inherits from `FileCloserBase` and provides an implementation for `closeFile()` to close a file given a file descriptor (see implementation in Listing 4.3). The client code in the `fileDemo()` method of Listing 4.4 opens a file, creates an instance of the `FileCloserSubclass` class, and invokes the `closeFile()` method on the instance through a base pointer. Because the base `closeFile()` method is declared as `virtual`, the invocation on line 12 of the client will resolve to the subclass, which safely closes the file. (Executing the program through Valgrind confirmed that no resources were leaked.) However, one static code analysis tool flagged a false positive "Leak" warning claiming the file descriptor was never closed.

```
1  // CWE675_Duplicate_Operations_on_Resource___open_81.h
2  // Pattern: file-close-virtual-method
3  #include <unistd.h>
4  #include <stdio.h>
5
6  namespace fileManager {
7    class FileCloserBase {
8    public:
9      virtual void closeFile(int fildes) {};
10   };
11
12   class FileCloserSubclass : public FileCloserBase {
13   public:
14     void closeFile(int fildes);
15   };
16 }
```

Listing 4.2: *File Close Virtual Method* false positive pattern header file.

Our documented changes to this pattern demonstrate how changes sometimes supplemented source code reduction to isolate the core structure causing the false positive. We observed the *File Close Virtual Method* pattern to be present in a situation where there was no class inheritance. Compared to the foregoing code example in Listings 4.2-4.4, there were two differences without inheritance: (1) the

```
1  // CWE675_Duplicate_Operations_on_Resource___open_81_goodG2B.cpp
2  // Pattern: file-close-virtual-method
3  #include "FileCloser.h"
4  #include <unistd.h>
5
6  namespace fileManager {
7    void FileCloserSubclass::closeFile(int fildes) {
8      close(fildes);
9    }
10 }
```

Listing 4.3: *File Close Virtual Method* false positive pattern implementation file.

```
1  // CWE675_Duplicate_Operations_on_Resource___open_81a.cpp
2  // Pattern: file-close-virtual-method
3  #include "FileCloser.h"
4  #include <fcntl.h>
5  #include <stdio.h>
6  #include <sys/stat.h>
7
8  namespace fileManager {
9    void fileDemo(void) {
10     int fildes = open("file.txt", O_RDWR|O_CREAT, S_IREAD|S_IWRITE);
11     FileCloserBase * fileCloser = new FileCloserSubclass();
12     fileCloser->closeFile(fildes);
13     delete fileCloser;
14   }
15   // Tool A FP: None
16   // Tool B FP: None
17   // Tool C FP: Leak. 'fildes' has gone out of scope and no
18   // longer references the resource of interest.
19 }
20
21 int main(void) {
22   fileManager::fileDemo();
23   return 0;
24 }
```

Listing 4.4: *File Close Virtual Method* false positive pattern client file.

`FileCloserSubclass` did not extend a user-defined base class, and (2) the file closer object in the client was declared as a pointer to the subclass type (so that the base class was not used). One change that did remove the false positive warning was removing the user-defined namespace `fileManager`. Indeed, having the `closeFile()` method defined in a separate namespace is a core structural feature of this pattern. Documenting such changes made the core structure of the false positive more explicit.

4.2.3 *Array Input Conditional* False Positive Pattern

A static code analysis tool sometimes flags a *negative* warning—that is, a warning that claim a particular event does *not* occur. A leak warning could potentially be a negative warning, because the warning claims a resource is never deallocated. However, in this context, we consider only warnings where the computer's resources suffer no adverse consequences if the event does not occur. For example, suppose a static code analysis tool flags a "variable is never used" warning in a program. Never using the variable may indicate a mistake or carelessness by a programmer, but the computer's resources would not incur leaks or other damage just because the variable is not used. Negative warnings are important to understand because they are easy to show to be false; we must give only one counterexample where the event actually does occur.

The *Array Input Conditional* false positive pattern in Listing 4.5 is one example of a program that gives a negative warning. On line 16 the `main()` function allocates a `char` array and passes it to a helper function to be initialized. The helper function reads data from standard input and appends the data to the array. The `main()` function then resumes execution to perform a search over the array for the search character. Upon finding a match, the program prints a message.

When analyzing this program, Tool C gave an "unreachable call" warning for the `printf()` statement, claiming the function is never invoked under any circumstances. Whether this call is actually invoked depends on the program's input—for

some inputs, the call will not be invoked, but for other inputs (namely, strings with the character `S` within the first nine positions), the call will be invoked. Thus, the `printf()` statement is necessary for program correctness. We therefore consider this "unreachable call" warning to be a false positive. Even in domains other than the Juliet test suite, programs may have edge cases that require special handling to ensure correct execution [33]. Claiming that the special sections are unreachable and the code is erroneous is unreasonable, in our view. These special sections are needed to correctly handle different inputs to the program and are dependent on the application logic.

4.3 Feedback from Static Code Analysis Tool Developers

As mentioned in Section 3.6, we validated false positive patterns in our catalog by requesting static code analysis tool developers to confirm the patterns to actually be false positives. We initially submitted the pattern *File Close Virtual Method* (see Section 4.2.2) to a developer of the commercial Tool C. The developer was initially inclined to dismiss the false positive pattern because the Juliet test suite is synthetic code. The developer, however, confirmed this pattern to be a false positive and recorded it to be fixed in a future release of the static code analysis tool.

The developer then requested the remaining false positive patterns from our catalog. We submitted 8 total patterns. Here are the responses we received from the tool vendor for the false positive patterns we submitted:

- 3 patterns were confirmed to be false positives. In two cases, the developer pointed to bugs in the tool models as producing the false positives. These patterns are *Buffer Store Socket*, *File Close Virtual Method*, and *Sscanf Uninit Var*.

- 1 pattern was characterized as a true positive. Although the warning could not arise in the calling context as written in the Juliet test case, the warning could

```
1  // CWE761_Free_Pointer_Not_at_Start_of_Buffer___char_console_62a.cpp
2  // Pattern: array-input-conditional
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #define SEARCH_CHAR 'S'
7  #define SIZE 10
8
9  void helper(char * &data) {
10   // Read from standard in and append to array.
11   size_t dataLen = strlen(data);
12   fgets(data+dataLen, (int)(SIZE-dataLen), stdin);
13 }
14
15 int main(void) {
16   char * data = (char *)malloc(SIZE*sizeof(char));
17   data[0] = '\0';
18   // Call helper function to initialize character array.
19   helper(data);
20   size_t i;
21   for (i=0; i < strlen(data); i++) {
22     if (data[i] == SEARCH_CHAR) {
23       // Tool A FP: None
24       // Tool B FP: None
25       // Tool C FP: Unreachable call. The following code will not execute
26       // under any circumstances.
27       printf("We have a match!\n");
28       break;
29     }
30   }
31   free(data);
32   return 0;
33 }
```

Listing 4.5: *Array Input Conditional* false positive pattern.

be a true positive in other calling contexts. This pattern is *Alloc Using External Size*.

- 1 pattern could not be reproduced, even when using the same static code analysis tool version. The tool vendor believed the reason was a difference in platform,

compiler version, or Standard Template Library (STL) version. The static code analysis tool that flagged the warning does not make assumptions about the STL version used; it analyzes the entire STL in the same manner as other code in the project. Thus, the tool's behavior is dependent on details other than the minimized source code. This pattern is *List Overrun*.

- We did not receive feedback regarding 3 patterns, which are *Array Input Conditional*, *Array Resize*, and *Clear List*.

4.4 False Positive Frequencies

Not only did we characterize false positives according to patterns, but we also measured their frequencies. Table 4.1 gives frequencies for the false positive patterns for the three static code analysis tools in our toolset. The false positive pattern families are listed in the first column. The table shows the number of times a particular false positive pattern appeared in the Juliet test suite for a given analysis tool.

In one sense, patterns with higher frequencies carry more significance because these patterns contribute more false positives to analysis reports that developers must sift through. These high frequencies demonstrate significant opportunity if we can harness the patterns to eliminate false positives. Some of the high frequencies, such as 3,322 for the *Predictable Conditional* pattern, are explained in that these patterns are composed of many variations. False positive patterns with low frequencies are still important to document, however. As we found with the two instances of the *File Close Virtual Method* pattern (see Section 4.3), a low-frequency pattern may still give insight to static code analysis tool developers for improving their products.

Due to the sheer number of some false positive patterns, we did not verify every warning instance. Instead, Table 4.1 reports the 95% confidence interval [34, 35] for the lower bound of the number of false positives. For example, we are 95% confident that Tool C flagged *at least* 240 instances of the *Alloc Using External Size* pattern in the Juliet test suite.

Table 4.1.: Frequencies of the false positive patterns from the Juliet test suite for our selected toolset.

False Positive Pattern	Tool A	Tool B	Tool C
1. Alloc using external size	0	0	240
2. Array input conditional	0	0	19
3. Array resize	0	0	12
4. Buffer store	0	0	428
5. Buffer underflow usage	3,685	0	0
6. Clear list	0	0	61
7. File close virtual method	0	0	2
8. List overrun	0	0	63
9. Null array access	0	55	0
10. Operation through alias	68	1,029	0
11. Predictable conditional	3,322	384	0
12. Return local param	1,456	0	0
13. Sscanf uninit var	0	0	2
14. Union simultaneous access	0	250	0

We obtained these frequencies through the following procedure:

1. For each false positive pattern, we isolated all warnings flagged by the static code analysis tool over Juliet that were of the same warning type dictated by the pattern. For example, the false warning for the *Alloc Using External Size* pattern is "uninitialized variable," so we isolated all "uninitialized variable" warnings from Tool C over Juliet. This set was a superset of the set of instances of the false positive pattern. That is, some warnings from this set might not have been instances of the pattern, but the set included all instances of the pattern.

2. We randomly sampled and inspected warnings from the set obtained in step 1. We determined the proportion of warnings from the sample that were instances of the false positive pattern in question.

3. We used our sampling results to compute a 95% confidence interval for the probability that a randomly chosen warning from the set was an instance of the false positive pattern. In general, we used a binomial probability calculator [34] that relied on the Clopper-Pearson (exact) method. If all of the sampled warnings were instances of the false positive pattern, we instead used the rule of three [36] from statistics to compute the confidence interval.

4. Finally, we applied the lower bound of the confidence interval to the remaining unsampled warnings to approximate the total number of instances of the false positive pattern [35].

It is worth noting that we applied pre-filtering to some false positive patterns in step 1 to reduce the number of warnings requiring sampling. In the context of the Juliet test suite, we pre-filtered the input warnings based on flow variants or CWEs. For example, we observed the *List Overrun* false positive pattern to be present only in flow variant 73 because this data flow variant uses a linked list data structure, a core structure of the pattern. The *List Overrun* pattern has a "buffer overrun" false warning. Therefore, as input to step 2 above, we isolated "buffer overrun" warnings that were flagged in test cases with flow variant 73. In general, it is possible our pre-filtering removed warnings that were actually instances of the false positive pattern in question. We minimized this chance by pre-filtering only when a false positive's structure was described by a small set of flow variants or CWEs. For instance, because the *List Overrun* false positive pattern uses a list as its core structure, and because flow variant 73 uses a list (and no other flow variants do), pre-filtering in this case did not degrade accuracy. Nevertheless, because we both pre-filtered warnings and reported lower bounds on the confidence intervals, the numbers reported in Table 4.1 are conservative. That is, the actual frequencies are likely to be higher.

Because we did not exhaustively inspect every warning from our codebase, our false positive catalog may not be complete for the Juliet test suite for our toolset. It is possible other false positive patterns exist that are not documented in the false positive hierarchy. However, we have shown that false positives occur as recurring patterns in our codebase.

In summary, we cataloged false positives from three static code analysis tools over the Juliet test suite into 14 core false positive pattern families. Chapter 6 proposes applying our same approach to identify false positive patterns in additional codebases, but for now, we have focused on the Juliet test suite for the reasons given in Section 3.2 on page 10. Based on our frequency data, we found that a large number of warnings from static code analysis tools could be characterized into a small number of patterns. We also inferred program structures, such as global variables used in conditions, that tended to generate false positives.

5 FALSE POSITIVE FILTERS

In Chapter 4, we showed that false positives in static code analysis tools occur as recurring patterns. Indeed, some patterns from our catalog occurred with frequencies on the order of thousands throughout the Juliet test suite. In this chapter, we present our preliminary results on harnessing these patterns to reduce the number of false positives flagged by static code analysis tools.

5.1 Filter Approach

To eliminate false positives, we implemented checkers that encoded the structures of the false positive patterns in order to identify the patterns in source code. We built the checkers using Clang's LibASTMatchers framework [37], which provides a domain-specific language for matching nodes in the source code's abstract syntax tree (AST). We chose this approach because the Clang AST provides a rich query interface with a syntax that is similar to C/C++ [38]. Appendix B gives more details of the LibASTMatchers framework through a simple example.

We used our pattern descriptions and reduced code snippets from the catalog to construct the checkers. For example, the description of the *Conditional Mem Leak* pattern (see Listing 4.1 on page 32 for the reduced code) states that memory is allocated dynamically outside a conditional, and the memory is freed in the `if` body of a conditional that tests a global variable. Therefore, we implemented a checker to identify a deallocation statement within the body of an `if` statement guarded by a global variable. Given a source code file, the checker then identified any occurrences of the pattern in the file along with the line number locations of the occurrences. There was a one-to-one correspondence between patterns and checkers; that is, we designed each checker to identify exactly one false positive pattern.

Figure 5.1 shows our approach to using a checker to filter out false positives in static code analysis tools. We created a script to automate the entire process in the figure for each warning message from a tool. First, we parsed the warning message for the filename generating the warning. Then we ran the checker over the source code file to identify any false positives of the same pattern for which the checker was designed. If the checker identified a false positive pattern on the warning line (*i.e.*, the same line flagged by the static code analysis tool), then the tool warning was predicted to be a false positive. Otherwise, if the checker did not identify the false positive pattern, or if the pattern occurred elsewhere in the source code file, the tool warning was predicted to be a true positive. Note that we use the term *checker* to refer to the program that checks for the presence of a false positive pattern. A *filter* is the script that applies a checker to eliminate false positives from the output of a static code analysis tool.

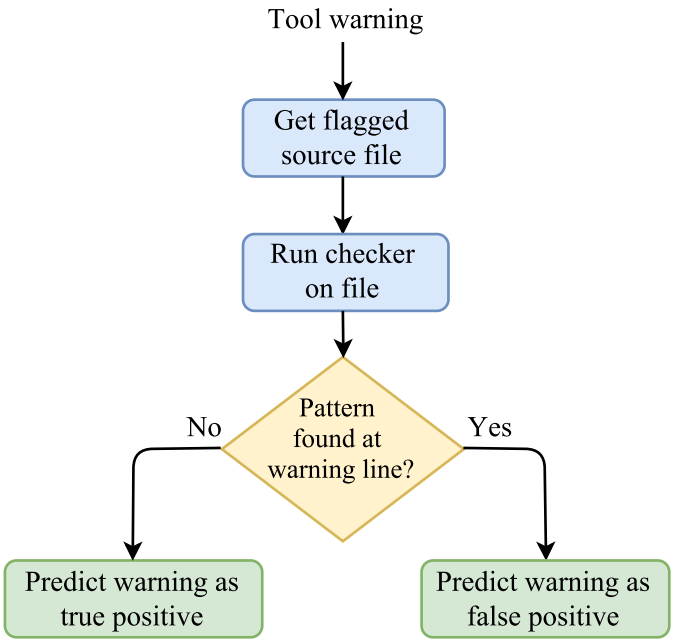

Figure 5.1.: Approach for applying checkers to filter out false positive warnings.

Note that we pre-filtered the set of tool warnings as input to the checker according to the warning class. For example, the false warning for the *Conditional Mem*

Leak pattern is "memory leak." Therefore, we did not apply the *Conditional Mem Leak* checker to predict "null pointer dereference" warnings, because predicting these warnings would require a different checker. We applied this *Conditional Mem Leak* checker to predict only "memory leak" warnings.

The foregoing discussion has been in the context of applying a single checker to predict a tool warning. It is possible for a checker to incorrectly predict a warning as true because the warning is a false positive of a different pattern. Thus, we believe multiple checkers should be applied to filter out as many false positives as possible. When multiple false positive checkers are applied, the warning is predicted as a false positive if any checker identifies the warning as a false positive. Likewise, a warning is predicted as a true positive if no checker identifies the warning as a false positive. Since each checker was designed to identify a separate false positive pattern, one checker could identify a warning as true while another checker could identify the same warning as false. Thus, when a warning is predicted to be a true positive, we can be confident only that it is not a false positive of the patterns for which we have applied checkers.

5.2 Preliminary Filter Results

As a preliminary experiment, we evaluated the following three false positive checkers for Tool A over the Juliet test suite: *Conditional Null Ptr*, *Return Local Param*, and *Conditional Mem Leak*. We selected these checkers for evaluation since they involved relatively simple structures. Results of the first two checkers are shown in the confusion matrices of Figures 5.2 and 5.3.

In each case, we ran the checker over all the warnings in the Juliet test suite for which the checker targeted. For example, we ran the *Conditional Null Ptr* checker over all 3,858 "null pointer dereference" warnings from Tool A in the Juliet test suite, since that is the false warning for this pattern. The "Predicted" columns in the figures indicate how many warnings the filter predicted as true positives and false positives,

		Predicted	
		True positive	False positive
Actual	True positive	26	0
	False positive	283	329
	Sample size	309	329
	Population	1,570	2,288

Figure 5.2.: Confusion matrix for *Conditional Null Ptr* checker for 3,858 "null pointer dereference" warnings from Tool A.

		Predicted	
		True positive	False positive
Actual	True positive	1	0
	False positive	0	307
	Sample size	1	307
	Population	1	1,526

Figure 5.3.: Confusion matrix for *Return Local Param* checker for 1,527 "pointer to local array variable returned" warnings from Tool A.

respectively. For the *Conditional Null Ptr* filter, 1,570 warnings were predicted to be true, while 2,288 were predicted to be false. Because these were large population sizes, we randomly selected a sample set from each population to give a 5% margin of error. For example, using an online sample size calculator [39], we determined we needed to sample 309 of the 1,570 warnings predicted as true positives to obtain a 5% margin of error. We manually inspected the sampled warnings to determine their actual labels and classified them according to the "True positive" and "False positive" rows. For instance, 26 of the "null pointer dereference" warnings predicted

as true positives were indeed true; 283 of the warnings predicted as true positives were actually false positives; and all warnings predicted as false were indeed false.

The terms *precision* and *recall* are helpful for describing the performance of our predictors. Precision is the portion of warnings predicted as false positives that really were false positives, while recall is the portion of false positives predicted as false positives [40]. We valued perfect precision, because we would rather miss filtering out a false positive than incorrectly filter out a true positive. In the latter case, the filter would wrongly instruct a software developer to ignore a warning message that actually indicated a flaw, thereby introducing defective software. Likewise, we valued high recall, although this was not a requirement. In other words, although filtering out all false positives would be ideal (*i.e.*, perfect recall), filtering out any false positives would be beneficial because developer workload would be reduced.

The precision and recall results for the *Conditional Null Ptr* and *Return Local Param* filters were promising. Both filters obtained perfect precision based on the instances sampled (*i.e.*, no true positives were filtered out). From these data and a binomial probability confidence interval calculator [34], we determined, with 95% confidence, that a warning predicted as false by a filter is indeed false with at least a 98.9% and 98.8% probability for the two filters, respectively. Therefore, a developer would be confident that no defects would be introduced by disregarding warnings predicted as false positives by these filters. On the other hand, recall was low (0.5376) for the *Conditional Null Ptr* filter because many of the false positives were not eliminated by the filter. In fact, our data show that for a warning predicted to be true by the filter, the probability that it is indeed true falls only between 5.6% and 12.1%, with 95% confidence [34]. Again, we are not alarmed at low recall, because low recall does not add any additional work for the developer in sifting through warning reports. Despite low recall, developer workload is still significantly reduced. For example, the *Conditional Null Ptr* filter reduced the total number of "null pointer dereference" warnings requiring developer inspection by 59.3%, while the *Return Local Param* filter reduced the total number of its corresponding warnings by 99.9%.

Figure 5.4 shows the results for the third filter, *Conditional Mem Leak*, to filter out "memory leak" false positives. Notice that these results are imprecise because 12% (86 out of 702) of the warnings filtered out as false positives were actually true positives. Because of the lower precision, we were not concerned with the population predicted as true positives, so we randomly sampled only 50 warnings from this population. Also, to obtain an accurate measurement, we inspected all 702 warnings that were predicted as false positives.

		Predicted	
		True positive	False positive
Actual	True positive	47	86
	False positive	3	616
	Sample size	50	702
	Population	3,685	702

Figure 5.4.: Confusion matrix for *Conditional Mem Leak* filter for 4,387 "memory leak" warnings from Tool A.

We observed that, of the 86 true positives that were incorrectly predicted as false, some instances conformed to a pattern. Earlier in this chapter we described the *Conditional Mem Leak* false positive pattern as memory allocated outside a conditional that tests a global variable, and the memory is freed later in the `if` statement's body. The pattern we observed among some of these true positives was that the memory pointer was deallocated, but the pointer no longer pointed to the beginning of the buffer. Therefore, memory was leaked, and the tool warning was a true positive.

To avoid filtering out true positives that are of this exception case, we implemented a specific checker to detect this case. We found this approach preferable to augmenting the original *Conditional Mem Leak* checker to become a single, monolithic checker that incorporated every exception case. We compare our approach to a general practitioner and a specialist in the medical field. The general practitioner assesses a wide variety

of situations, but the specialist focuses on a specific area with higher accuracy. In the same way, the *Conditional Mem Leak* checker broadly predicted warnings based on a general pattern, while the specific checker refined the predictions based on a specific pattern. To increase precision, then, we ran the specific filter on the warnings predicted to be false positives by the general filter. In this way, the specific checker can be considered a "second-level checker" since it filters warnings a second time.

Figure 5.5 gives the new results after applying the specific checker. Warnings previously predicted as true did not change their predictions, since the specific checker did not filter these. We did not re-sample from this population, so there are no data to report in the first column. 48 warnings previously predicted as false positives were now correctly predicted as true positives, all of which we verified by hand. Through the use of the specific checker, we increased precision from 87.75% to 94.19%.

		Predicted	
		True positive	False positive
Actual	True positive	--	38
	False positive	--	616
	Sample size	--	654
	Population	3,733	654

Figure 5.5.: Confusion matrix for *Conditional Mem Leak* filter after applying the specific checker.

The remaining 38 warnings were incorrectly predicted as false positives because of a different reason: there were two "memory leak" warnings on the same line for two different variables—one warning a false positive and the other a true positive. The checker correctly identified the false positive pattern for the first warning. However, because our filtering mechanism did not distinguish between warnings for different variables, the checker incorrectly predicted the second warning as false since it was

on the same line. This error could be resolved by modifying the checkers to output the variable name. Because this component of the thesis is preliminary work, we left this improvement for future work.

In summary, our preliminary results for false positive filters show that we were able to harness our cataloged patterns to reduce the number of false positives that the developer must sift through. Our data indicated high precision, meaning that when a checker filtered out a warning as a false positive, we were confident it could safely be ignored. Even when a general checker filtered out true positives, we were able to design a specific checker to refine the results, increasing precision. When applying three filters, we were able to eliminate 59.3%, 99.9%, and 14.0%, respectively, of the warnings as false positives, while having a minimum precision of 94.19%.

6 CONCLUSION AND FUTURE WORK

In this thesis, we presented an approach for identifying false positive patterns in static code analysis. Our catalog of 14 distinct false positive patterns showed that a large number of warnings in the Juliet test suite could be characterized into a small number of patterns. We tested both open-source and commercial static code analysis tools and observed some patterns to be common across tools. We showed that it is possible to infer program structures that tend to cause false positives, such as global variables used in conditions. We also demonstrated the practicality of using a standardized test suite in helping static code analysis tool developers discover bugs in their tools. Lastly, we implemented checkers to identify the structures of false positive patterns to filter out false positive warnings from static code analysis tools. High precision of our initial results indicated this approach to be a promising false positive elimination technique that merits further research.

Some areas for future work are listed below:

- **Expand toolset.** We selected two open-source static code analysis tools and one commercial tool for our study. This toolset was sufficient to demonstrate the presence of false positive patterns, but additional tools would be needed to better study the existence of false positive patterns across multiple tools or to compare open-source and commercial tools.

- **Expand codebase.** We realize the Juliet test suite may not be representative of real code, so we plan to test our approach on open-source libraries, such as Coreutils [41] or POCO [42]. Whether the same false positive patterns (or new patterns) emerge remains to be seen.

- **Automate source code reduction.** Currently, our approach involves a large amount of manual work, which will hinder our ability to scale to larger code

bases. A potential candidate for automation is in reducing the false positive code snippets to their minimized forms. Iterative techniques, such as delta debugging [43], have been used to reduce code snippets that generate compiler warnings. However, reducing code snippets that generate static code analysis warnings is more challenging because of the difficulty of classifying a warning as true or false. Thus, it is possible for an automated technique to reduce too far and convert a false positive warning into a true positive. Nevertheless, automated reduction is still an important research area.

- **Explore false positive abstraction levels.** We indicated that we observed both implementation-level and design-level false positive patterns in our catalog. We plan to explore this distinction further to better understand the impact of abstraction levels on encoding the pattern structures.

- **Specify metamodel for describing patterns.** We have specified false positive patterns in terms of both informal English descriptions and low-level, minimized code snippets. To eliminate false positives, our approach requires the structures of the false positive patterns to be encoded. To simplify the process for additional patterns, we plan to develop a language-independent metamodel for expressing patterns. Doing so would allow us to separate the representation of the pattern from the checker logic itself. We envision a generic checker that would take as input both the pattern representation and a source file to inspect for the pattern.

We have included an abbreviated catalog of our false positive patterns in Appendix A. The full catalog can be accessed at the following location: `https://github.com/SEDS/mangrove-catalog/wiki`.

REFERENCES

REFERENCES

[1] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[2] Martin Fowler and Matthew Foemmel. Continuous integration. `https://martinfowler.com/articles/continuousIntegration.html`, 2006. Accessed: 2017-07-14.

[3] Panagiotis Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.

[4] The MITRE corporation. `https://www.mitre.org/`. Accessed: 2017-07-12.

[5] Common Weakness Enumeration frequently asked questions. `http://cwe.mitre.org/about/faq.html`. Accessed: 2016-09-13.

[6] Wikipedia. List of tools for static code analysis — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis&oldid=739038439`, 2016. Accessed: 2016-09-13.

[7] Microsoft code analysis tool .NET (CAT.NET) v1 CTP - 32 bit. `https://www.microsoft.com/en-us/download/details.aspx?id=19968`. Accessed: 2016-09-18.

[8] Findbugs - find bugs in Java programs. `http://findbugs.sourceforge.net/`. Accessed: 2016-09-19.

[9] A look at CWE coverage across open source and commercial static analysis tools. `https://codedx.com/a-look-at-cwe-coverage-across-open-source-and-commercial-static-analysis-tools/?v=7516fd43adaa`. Accessed: 2016-09-13.

[10] Markus Mock. Dynamic analysis from the bottom up. In *WODA 2003 ICSE Workshop on Dynamic Analysis*, page 13, 2003.

[11] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

[12] Muhammad Nadeem, Byron J. Williams, and Edward B. Allen. High false positive detection of security vulnerabilities: A case study. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 359–360. ACM, 2012.

[13] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of SQL injection vulnerabilities in web services. In *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*, pages 301–306. IEEE, 2009.

[14] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM, 2004.

[15] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM.

[16] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *International Static Analysis Symposium*, pages 295–315. Springer, 2003.

[17] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 762–774. ACM, 2014.

[18] Software assurance reference dataset. `https://samate.nist.gov/SRD/testsuite.php`. Accessed: 2016-09-19.

[19] Lakshmi Manohar Rao Velicheti, Dennis C. Feiock, Manjula Peiris, Rajeev Raje, and James H. Hill. Towards modeling the behavior of static code analysis tools. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, CISR '14, pages 17–20, New York, NY, USA, 2014. ACM.

[20] U. Yuksel and H. Sozer. Automated classification of static code analysis alerts: A case study. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 532–535, Sept 2013.

[21] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 161–170. IEEE, 2009.

[22] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 35–42. ACM, 2017.

[23] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, pages 1–6. ACM, 2015.

[24] T. Muske, A. Datar, M. Khanzode, and K. Madhukar. Efficient elimination of false positives using bounded model checking. In *VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle*, pages 13–20, 2013.

[25] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 470–481. IEEE, 2016.

[26] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[27] David Svoboda. Is Java more secure than C? `https://insights.sei.cmu.edu/sei_blog/2015/10/is-java-more-secure-than-c.html`, Oct 2015. Accessed: 2017-06-14.

[28] The Common Weakness Enumeration (CWE) Initiative, MITRE Corporation. `http://cwe.mitre.org/`. Accessed: 2017-07-12.

[29] Jun Wang, Mingyi Zhao, Qiang Zeng, Dinghao Wu, and Peng Liu. Risk assessment of buffer "heartbleed" over-read vulnerabilities. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 555–562. IEEE, 2015.

[30] Valgrind. `http://valgrind.org/`. Accessed: 2016-09-19.

[31] James O. Coplien. Software design patterns. In *Encyclopedia of Computer Science*, pages 1604–1606. John Wiley and Sons Ltd., Chichester, UK, 2003.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented programming, 1995.

[33] Josh Zimmerman and Jonathan Clark. Unit testing. `https://www.cs.cmu.edu/~rjsimmon/15122-s13/rec/07.pdf`, 2012. Accessed: 2017-06-27.

[34] Daniel S. Soper. Binomial probability confidence interval calculator. `http://www.danielsoper.com/statcalc/calculator.aspx?id=85`, 2017. Accessed: 2017-03-20.

[35] Jyoti Sarkar. Personal communication.

[36] Borko D. Jovanovic and Paul S. Levy. A look at the rule of three. *The American Statistician*, 51(2):137–139, 1997.

[37] Matching the Clang AST. `https://clang.llvm.org/docs/LibASTMatchers.html`, 2017. Accessed: 2017-06-26.

[38] Introduction to the Clang AST. `http://clang.llvm.org/docs/IntroductionToTheClangAST.html`, 2017. Accessed: 2017-06-26.

[39] Sample size calculator. `https://www.surveysystem.com/sscalc.htm`, 2012. Accessed: 2017-04-18.

[40] Michael Buckland and Fredric Gey. The relationship between recall and precision. *Journal of the American society for information science*, 45(1):12, 1994.

[41] Coreutils - GNU core utilities. `https://www.gnu.org/software/coreutils/coreutils.html`. Accessed: 2017-06-29.

[42] POCO C++ libraries. `https://pocoproject.org/`. Accessed: 2017-06-29.

[43] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, pages 142–151. ACM, 2006.

APPENDICES

A FALSE POSITIVE CATALOG

This appendix is an abbreviated catalog briefly describing each false positive pattern cataloged in this research study. The false positive pattern names, warnings, descriptions, and tools are included below. The 14 core pattern families are numbered with Arabic numerals in the list and are sorted in alphabetical order by the pattern name. Child patterns are indented to the right under their respective parent patterns. Pattern names are set in italic font throughout this catalog. Names of abstract patterns (*i.e.*, patterns with child patterns) are followed by the text "Abstract Pattern." Our complete catalog with further details is made available on GitHub at the following location: `https://github.com/SEDS/mangrove-catalog/wiki`.

1. *Alloc Using External Size* Abstract Pattern

   Common warning: Uninitialized variable

   Common description: Memory is allocated dynamically. The size of memory allocation is determined based on a variable that is initialized outside the function that allocates memory, and the value of this size variable is somehow passed into the function. Inside the function, memory is (1) allocated using the size value, (2) initialized by making use of the same size value, and (3) accessed later within the same function that initializes it.

   (a) *Alloc Using Function Param*

   Description: The value to determine the size of memory allocation is passed from one function to another using function parameters. The memory is accessed later within the function that allocates the memory.

   Tools: Tool C

(b) *Alloc Using Global Var*

Description: The value to determine the size of memory allocation is passed from one function to another using a static global variable. The memory is accessed later within the same function. (The pattern is verified to exist when using `int`, `char`, `float`, and `double` data types.)

Tools: Tool C

(c) *Alloc Using Property*

Description: The value to determine the size of memory allocation is a private property that is initialized in the class's constructor. (The property is never modified outside the constructor.) The class destructor contains the code that performs memory allocation, initialization, and access.

Tools: Tool C

2. *Array Input Conditional*

Warning: Unreachable call

Description: A character array is allocated and sent to a helper function to be initialized. (In the code snippet in the catalog, the array is initialized with input from the user through `fgets()`.) The array is then traversed in a `for` loop and an action is performed if the array contains a character of interest.

Tools: Tool C

3. *Array Resize*

Warning: Useless assignment

Description: A string is allocated, initialized, and passed to a helper function. The helper function (1) saves the initial length of the string to a variable, (2) modifies the string (using `fgets()` in the example in the catalog), and (3) saves the new length to the same variable.

Tools: Tool C

4. *Buffer Store* Abstract Pattern

   Common warning: Tainted buffer access

   Common description: Data is written into a buffer, and the amount of data written is limited by taking into account both (1) the size of the buffer and (2) the amount of space already used in the buffer.

   (a) *Buffer Store Fgets*

   Description: Data is read from the user through `fgets()` and written to the buffer. Afterwards, the newline character appended to the buffer by `fgets()` is replaced with a null character.

   Tools: Tool C

   (b) *Buffer Store Socket*

   Description: Data is read from a socket and written to a buffer. Afterwards, a null character is appended to the end of the data.

   Tools: Tool C

   (c) *Buffer Store Strncat*

   Description: Data is read from an environment variable and written to a buffer using `strncat()`.

   Tools: Tool C

5. *Buffer Underflow Usage*

   Warning: Buffer is accessed out of bounds

   Description: A value is stored to a buffer after the buffer is initialized. Let L be the length of the value that is stored. The buffer must be initialized to a value of length less than L - 1.

   Tools: Tool A

6. *Clear List*

   Warning: Use after free

   Description: Non-zero integer values are pushed onto a list, and the values are read using an iterator. If any value is zero, the list is cleared during iteration.

   Tools: Tool C

7. *File Close Virtual Method*

   Warning: Leak

   Description: A file is opened with `open()` and closed by passing the file descriptor to a virtual method of a class defined in a user-defined namespace. (A more detailed description is available in the online catalog.)

   Tools: Tool C

8. *List Overrun*

   Warning: Buffer overrun

   Description: A list is initialized with at least one element and passed to another function, where it is accessed. An "access" includes the `front()` and `back()` methods but not `pop_front()`, `pop_back()`, or `begin()`. (These latter methods do not produce a FP.)

   Tools: Tool C

9. *Null Array Access*

   Warning: Value is garbage or undefined

   Description: A character array is allocated, the first value is assigned the null char, and then every character in the array (up to its length) is accessed.

   Tools: Tool B

10. *Operation Through Alias* Abstract Pattern

    Common description: An operation is performed on a resource through an alias.

(a) *Ref Ptr Read*

Warning: Stored value is never read

Description: A character array is allocated on the stack and read through a reference to a pointer.

Tools: Tool B

(b) *Subscope Leak*

Warning: Memory leak

Description: A pointer is allocated memory, and it is freed through a different pointer of the same name in a subscope. (By *subscope*, we mean a scope distinct from the allocation's scope in which the variable that has been allocated memory is still visible.)

Tools: Tool A

11. *Predictable Conditional* Abstract Pattern

Common description: A conditional's truth or falsity can always be predicted. For example, a conditional may test a global whose value never changes.

(a) *Conditional Mem Leak*

Warning: Memory leak

Description: Memory is allocated dynamically before a conditional that tests a global variable, and the memory is freed in the `if` body of the conditional.

Tools: Tool A, Tool B

(b) *Conditional Null Ptr*

Warning: Possible null pointer dereference

Description: A null variable `v` is set to a value inside the `if` body of a conditional with a global variable that always evaluates to true, and `v` is used later.

Tools: Tool A, Tool B

(c) *Conditional Uninit Var*

Warning: Uninitialized variable

Description: An uninitialized variable `v` is set to a value inside the `if` body of a conditional with a global variable that always evaluates to true, and `v` is used later.

Tools: Tool A, Tool B

(d) *Externally Defined Symbols* Abstract Pattern

Common description: A conditional uses an externally defined symbol that the analysis tool cannot reason with.

i. *Conditional Mem Leak External Function*

Warning: Memory leak

Description: Memory is allocated dynamically, and it is deleted in the `if` body of a conditional with boolean function (defined externally) that always evaluates to true.

Tools: Tool A, Tool B

ii. *Conditional Mem Leak External Var*

Warning: Memory leak

Description: A pointer is allocated memory on the heap (outside the conditional), and the memory is freed in the `if` body of a conditional that tests a `const` global variable that is defined externally.

Tools: Tool A, Tool B

iii. *Conditional Null Ptr External Function*

Warning: Null pointer dereference

Description: A null pointer is initialized in the `if` body of a conditional with a boolean function, and the pointer is dereferenced later.

Tools: Tool A, Tool B

iv. *Conditional Uninit Var External Function*

Warning: Uninitialized variable / undefined pointer dereference

Description: An object is allocated in the `if` body of a conditional with a boolean function, and the object is accessed later.

Tools: Tool A, Tool B

v. *Conditional Uninit Var External Var*

Warning: Uninitialized variable / undefined pointer dereference

Description: A variable `v` is initialized in the `if` body of a conditional that tests a `const` variable (that evaluates to true) defined externally, and `v` is accessed later outside the `if` statement.

Tools: Tool A, Tool B

vi. *Intermediate Function Call*

Warning: Undefined pointer dereference

Description: An externally-defined function is invoked between reads of a global variable that is evaluated in a pair of conditionals. The externally-defined function must not modify the global variable. Control flow occurs through the conditionals such that the undefined pointer dereference is logically impossible. (The analysis tool assumes the externally-defined function modifies the global variable so that the undefined pointer dereference occurs.)

Tools: Tool B

12. *Return Local Param*

Warning: Pointer to local array variable returned

Description: A function with a pointer parameter returns the pointer to the caller and does not modify it.

Tools: Tool A

13. *Sscanf Uninit Var*

    Warning: Uninitialized variable

    Description: The `sscanf()` C library function is used to read a value from a literal string, and the read will succeed given the literal string and the data type(s) being read.

    Tools: Tool C

14. *Union Simultaneous Access*

    Warning: Memory leak

    Description: Memory is allocated to one union member and freed through a different member.

    Tools: Tool B

B CLANG CHECKER EXAMPLE

This appendix describes the LibASTMatchers framework [37] through an example. In order to filter out false positives from static code analysis tools, we used the LibASTMatchers framework to build checkers for identifying false positive structures.

We selected the *Return Local Param* false positive pattern from our catalog for this discussion because of its simple structure.[1] Listing B.1 gives the reduced code snippet for the *Return Local Param* pattern. The `main()` function allocates a buffer on the stack and passes a pointer to the buffer into a helper function. The helper function merely returns the pointer directly to the caller. Tool A warns that the helper function returns a pointer to a local array variable. However, there is no flaw, because the pointer was defined in the caller function and therefore has meaning to the caller.

To build a Clang checker to identify this pattern, we needed to encode the pattern's structure. The key structural component of this pattern is the return of a pointer variable that was passed into the function. To understand how to encode this structure, we found it helpful to view the source code's abstract syntax tree (AST). Figure B.1 shows the relevant portion of Clang's AST of the program given in Listing B.1.

Reading the Clang AST in Figure B.1 is fairly intuitive. Top-level nodes appear at the left-hand side, while nodes further down in the tree are indented to the right. Lines 1-6 and 7-25 give the structures of the `helper()` and `main()` functions, respectively. The two `CompoundStmt` nodes on lines 3 and 8 represent the two function blocks designated with braces in the source code. The five source-code level statements in the `main()` function are represented by the AST nodes beginning on lines 9, 11, 13,

[1]The *Return Local Param* pattern was described in Section 3.4, and results for applying the *Return Local Param* filter to eliminate false positives were given in Figure 5.3 on page 45.

```c
// CWE665_Improper_Initialization___char_cat_21.c
// Pattern: return-local-param

char * helper(char * data) {
  // Tool A FP: Pointer to local array variable returned.
  // Tool B FP: None
  // Tool C FP: None
  return data;
}

int main(void) {
  char * data;
  char dataBuffer[10];
  data = dataBuffer;
  data = helper(data);
  return 0;
}
```

Listing B.1: Reduced code snippet for *Return Local Param* false positive pattern.

```
1  |-FunctionDecl 0x406fe20 <return-local-param.c:4:1, line:9:1> line:4:8 used helper 'char *(char *)'
2  | |-ParmVarDecl 0x406fd58 <col:15, col:22> col:22 used data 'char *'
3  | `-CompoundStmt 0x406ff68 <col:28, line:9:1>
4  |   `-ReturnStmt 0x406ff50 <line:8:5, col:12>
5  |     `-ImplicitCastExpr 0x406ff38 <col:12> 'char *' <LValueToRValue>
6  |       `-DeclRefExpr 0x406ff10 <col:12> 'char *' lvalue ParmVar 0x406fd58 'data' 'char *'
7  `-FunctionDecl 0x4070050 <line:11:1, line:17:1> line:11:5 main 'int (void)'
8    `-CompoundStmt 0x40bbe78 <col:16, line:17:1>
9      |-DeclStmt 0x40bbb70 <line:12:5, col:16>
10     | `-VarDecl 0x40bbb10 <col:5, col:12> col:12 used data 'char *'
11     |-DeclStmt 0x40bbc68 <line:13:5, col:24>
12     | `-VarDecl 0x40bbc08 <col:5, col:23> col:10 used dataBuffer 'char [10]'
13     |-BinaryOperator 0x40bbce8 <line:14:5, col:12> 'char *' '='
14     | |-DeclRefExpr 0x40bbc80 <col:5> 'char *' lvalue Var 0x40bbb10 'data' 'char *'
15     | `-ImplicitCastExpr 0x40bbcd0 <col:12> 'char *' <ArrayToPointerDecay>
16     |   `-DeclRefExpr 0x40bbca8 <col:12> 'char [10]' lvalue Var 0x40bbc08 'dataBuffer' 'char [10]'
17     |-BinaryOperator 0x40bbe18 <line:15:5, col:23> 'char *' '='
18     | |-DeclRefExpr 0x40bbd10 <col:5> 'char *' lvalue Var 0x40bbb10 'data' 'char *'
19     | `-CallExpr 0x40bbdd0 <col:12, col:23> 'char *'
20     |   |-ImplicitCastExpr 0x40bbdb8 <col:12> 'char *(*)(char *)' <FunctionToPointerDecay>
21     |   | `-DeclRefExpr 0x40bbd38 <col:12> 'char *(char *)' Function 0x406fe20 'helper' 'char *(char *)'
22     |   `-ImplicitCastExpr 0x40bbe00 <col:19> 'char *' <LValueToRValue>
23     |     `-DeclRefExpr 0x40bbd60 <col:19> 'char *' lvalue Var 0x40bbb10 'data' 'char *'
24     `-ReturnStmt 0x40bbe60 <line:16:5, col:12>
25       `-IntegerLiteral 0x40bbe40 <col:12> 'int' 0
```

Figure B.1.: Clang AST of the *Return Local Param* program in Listing B.1.

17, and 24. Finally, variable declarations are represented by `DeclStmt` nodes (as on lines 9 and 11) with a single child node, while assignment statements are represented

by `BinaryOperator` nodes (as on lines 13 and 17) with two child nodes, one for either side of the assignment operator. For a deeper understanding, we suggest studying Clang's introduction to the AST [38].

Listing B.2 gives the LibASTMatchers code snippet we created to identify the structure of the *Return Local Param* pattern. Note that the snippet is purely declarative; we merely describe what the structure is, not how to identify it. By comparing the code snippet in Listing B.2 with the AST in Figure B.1, the code snippet becomes understandable. We start on line 2 by matching a return statement, so this checker will identify the location of the return statement used in the false positive pattern. Next, we need to extract the variable being returned, so lines 3-5 check that the return statement node has a `DeclRefExpr` child node (*i.e.*, a reference to a declared variable). Because of the `ignoringParenImpCasts` component, any implicit casts or parenthesis are ignored in this match for a child. Finally, lines 6-8 check that the variable being returned refers to a function parameter that is of a pointer type. The `bind` statements assign user-defined labels to AST nodes that can be retrieved later in the match handler when the checker fires. This allows, for instance, our checkers to lookup and print line numbers of pattern matches.

```
1  StatementMatcher returnMatcher =
2    returnStmt(
3      has(
4        ignoringParenImpCasts(
5          declRefExpr(
6            to(
7              parmVarDecl(
8                hasType(isAnyPointer())
9              ).bind("param_var")
10           )
11         )
12       )
13     )
14   ).bind("return_stmt");
```

Listing B.2: Declarative snippet using Clang's LibASTMatchers framework to encode the *Return Local Param* pattern's structure.