ABSTRACT

Title of dissertation: SOURCE CODE REDUCTION
TO SUMMARIZE FALSE POSITIVES

Matías Marenchino, Master of Science, 2015

Dissertation directed by: Professor Adam Porter
Department of Computer Science

The main disadvantage of static code analysis tools is the high rates of false positives they produce. Users may need to manually analyze a large number of warnings, to determine if these are false or legitimate warnings, reducing the benefits of automatic static analysis.

Our long term goal is to significantly reduce the number of false positives that these tools report. A learning system could classify the warnings into true positives and false positives by means of features extracted from the program source code. This work implements and evaluates a technique to reduce the source code producing false positives into code snippets that are simpler to analyze.

Results indicate that the method considerably reduces the source code size and it is feasible to use it to characterize false positives.

SOURCE CODE REDUCTION TO SUMMARIZE FALSE POSITIVES

by

Matías Marenchino

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2015

Advisory Committee:
Professor Adam Porter, Chair/Advisor
Professor Alan Sussman
Professor Rance Cleaveland

Acknowledgments

Thanks to my advisor, Professor Adam Porter, for giving me the chance to work on this interesting project and for his guidance in completing this research. I would also like to thank Professor Jeff Foster who gave me helpful advice and Professor James Hill that provided me with some tools too.

I would like to acknowledge my thesis committee, Professor Alan Sussman and Professor Rance Cleaveland, for taking the time to review this work.

Thanks to my friends, my family and my wife. None of this would have been possible without your trust in me and support.

Table of Contents

List of Figures

List of Abbreviations

DD	Delta Debugging
DDT	Delta Debugging with topformflat integration
FN	False Negative
FP	False Positive
IRD	Iterative Reduction Debugging
IRDT	Iterative Reduction Debugging with topformflat integration
LOC	Source Lines of Code
TP	True Positive
SCA	Static Code Analysis
SCATE	Static Code Analysis Tool Evaluator

Chapter 1: Introduction

The process of testing consists of evaluating whether an activity or a product has been developed correctly. Software testing refers to this process performed to support quality assurance of a program under test. In order to test a system, different techniques are used. These, can be divided into two categories: dynamic analysis and static analysis. The former type of testing requires the execution of the software under test. The latter, which we are going to focus on, consists of analyzing the source code or object code to detect errors or flaws.

These two techniques of verification have their advantages and disadvantages. The dynamic execution of tests is relative easy and can be used to measure if the system satisfies its requirements. Also, executing the software in the runtime environment where it will be running could provide confidence that it will work as expected. On the other hand, by dynamic analysis, we can only show the existence of faults in the code. Even in the case that we have executed several tests on the system, we cannot guarantee the absence of vulnerabilities.

Static code analysis (SCA) is a more exhaustive approach to detect faults. Tools performing this analysis usually scan the whole code to identify flaws. These tools check the compliance of specific rules and detect instances of vulnerabilities

patterns. Due to the low cost of implementation, SCA tools are extremely useful to software verification. The main limitation of SCA tools is that they produce plenty of warnings, many of them are false alarms. These could be false positives or some other issues that are informed to the users, but are not faults. In [1], Lobo de Mendonça et al. do a literature review and detail the false positive rate reported in many publications. This rate is in general over 20%.

The primary complication related to false positives is that the user or software developer should manually inspect every single warning in order to determine its truthfulness. This requires time and effort and makes it an expensive process. Additionally, the large amount of false positives tend to cause confusion to developers which may ignore the SCA tool warnings. This could lead to the omission of true positives and important bugs and flaws. For instance, in a project of a Computer Science introductory undergraduate course, the build automation system generated a report by means of a SCA tool. Students were supposed to use this tool to help them in the development process but many of them did not pay attention to it, because of the abundance of false positives.

There are many techniques used today to reduce the number of false positives reported by SCA tools [2]. These could be as elementary as turning off some specific types of checks or changing the source code not to produce the warning. But some SCA tools provide more sophisticated features like processing annotations that indicate the location of false positives, so that the warning is ignored the next time the tool generates a report, or using statistical information to detect likely false positives. Even though some of these techniques could be automatic, many of them

are manual and require the experience of the software developers.

The report [3] presents another way to provide insights of user practice regarding warnings and false positives. This survey revealed that most of the users are concerned about high priority warnings. But, surprisingly, a large number of users also take into account low priority warnings. In general, low priority reports are less accurate and, therefore, they tend to have a higher rate of false positives.

To the best of our knowledge, very little work has been published on warning classification and false positives detection. An interesting idea is presented in [4]. Google's code base was used to build models to predict if a warning is a false positive or not. A logistic regression model was used to determine how accurate a warning is. The logistic regression factors are obtained from complexity metrics, the change history of files, the programming language, the description of the SCA tool warnings. The main problem of this approach is that many external factors are considered. These may not be available in an initial release of a program, like the change history of files. Additionally, the source code is not analyzed in detail to predict false positives, just by means of complexity metrics.

This thesis aims to help in the process of reducing the number of false positives reported by SCA tools. To this end, we have to characterize the false positives produced by the tools and then learn from them, so as to infer whether a warning is a true or false positive. Besides the source code metadata obtained in [4], we strongly believe that the warnings could be effectively predicted if we analyze the source code in detail. In order to make the analysis simpler, the source code has to be reduced or summarized.

We will be using open source projects and the Juliet Test Suite [5] to evaluate our approach. Juliet is a collection of programs with identified flaws. The code contains functions named as "good" and "bad" that help to recognize if a detected vulnerability is in fact a flaw (or true positive) or it is a false positive. We ran SCA tools on this projects to get a large number of false positives. To obtain a simplified representation of each false positive, we reduce the original source code file into code snippets producing exactly the same false positive. These code snippets are obtained by isolating failure causes by modified versions of Delta Debugging [6] on the lines of the initial source code. It is worth pointing out that the snippets result in valid programs, i.e., programs that can be compiled by the way the original code was compiled.

Besides being a simple representation of the false positive, the code snippets are source code. Not bytecode or any other abstract intermediate representation of the code. This can really help a software developer in his process to understand the reason of a warning message. In fact, by examining the code snippets we were able to detect problems in the SCA tools. For example, we found that a SCA tool generates a false positive if a global variable is used in a condition, but it does not report a warning if the variable is replaced by its actual value. The process of source code inspection of these snippets is straightforward: an original source code containing thousand of LOC may be reduced in a snippet having just dozens of lines.

Once we obtain a simplified version of the code generating a warning, analyzing it should become a simpler task. In this work we focus on the problem of the reduction of the source code into the snippets.

Chapter 2: Background

This chapter presents background information for the remaining of this work. First we introduce SCA tools and the notion of false positives in binary classification, then we describe the Delta Debugging algorithm, which results in one of the key aspects of this work. Finally we introduce Iterative Reduction Debugging, another method to minimize sets.

2.1 Static Code Analysis tools

Static code analysis [7] is the analysis of computer software performed statically, without executing the program, to find vulnerabilities on it. Usually, the source code of the program or the object code are analyzed.

The kind of tools that perform this analysis are usually called static code analysis tools or SCA tools. These automatic tools should, ideally, detect flaws with a high degree of confidence that what is detected is actually a flaw. In practice, many of the SCA tools help the programmers to find potential weaknesses in the code more efficiently, but the process of flaw detection is far from being completely automatic. And the primary reason is the presence of a large amount of False Positives (FP).

A false positive is an error in the process of evaluation in which a test result mistakenly indicates the presence of a condition while in reality it is not the case. In binary classification, we can also define true positives, false positives and true negatives. Table 2.1 introduces them for the domain of SCA tools results.

	SCA tool reports a flaw	SCA tool does not report a flaw
Actual flaw	True Positive	False Negative
Not a flaw	False Positve	True Negative

Table 2.1: Binary classification data reporting

There are different techniques to conduct static code analysis. Many of them are included in [8] and [9], we describe some of them briefly:

- Data Flow Analysis: This is a process to obtain run-time information of the software in a static manner: without actually executing the code. This analysis makes use of the control flow graph [10]: an abstract representation of all paths that could be traversed in a program while it is executed. Every node in this graph is a basic block, which is a portion of the code containing a single entry point and also a single exit point. The paths from basic block to another is represented by directed edges in the graph. The data relationship of a program is done by means of the control flow graph. Additionally, it helps to identify portions of code to be optimized.

- Taint Analysis: The technique aims to identify variables that are tainted with user input. Then, it traces these variables to find vulnerable functions.

- Symbolic Analysis: The method is conceived to reason about non-constant

variables values. It generates a formal mathematical characterization of the calculations.

2.2 Delta Debugging

Delta Debugging (DD) is a technique to automatically minimize a failure-inducing program input in order to obtain its minimal subset which produces exactly the same original failure. It makes use of a intuitive method to minimize: binary search. We will first present an intuitive example and then we formally define it.

Suppose an electronic system uses a small keyboard as depicted in figure 2.1. Consider that the system fails if we press all the eight buttons in the keyboard at the same time. But, actually, the failure is produced only when pressing the keys labeled 'A', 'B' and 'F', but we do not know that yet. If we try to find what is the minimum combination of keys producing the error, we could proceed as follows.

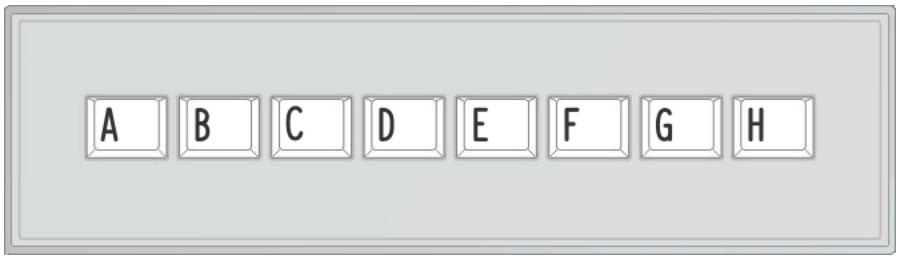

Figure 2.1: DD example: 8 buttons keyboard

Pressing no button does not reproduce the error, so we divide the keys into two sets: from 'A' to 'D' and from 'E' to 'H'. Pressing all the keys in these subsets does not produce the error, so we divide into two again, with the subsets $S_1 = \{A, B\}$, $S_2 = \{C, D\}$, $S_3 = \{E, F\}$ and $S_4 = \{G, H\}$. Now we consider these sets

individually and their complements. Pressing the buttons in S_i does not fail for any i. But, if we consider the complements, S_2, S_3, S_4 does not produce the failure, but S_1, S_3, S_4 does. So, we know that a subset of $S_1 \cup S_3 \cup S_4$ should fail.

We can try to remove S_3 from it, with no succes. Then, if we remove S_4, we get $S_1 \cup S_3$ which also fails. Now we can divide the subsets, getting single elements: $\{A, B, E, F\}$. We proceed by removing single elements: $\{B, E, F\}$ and $\{A, E, F\}$ do not fail, but $\{A, B, F\}$ does. The last step is simply to remove single elements from it, but they will not fail. Therefore, we have found the minimum failing subset: $\{A, B, F\}$.

We could use a diagram to represent the technique for the keyboard example, as in figure 2.2. To make the diagram complete, many tests that were already done before are included. In practice, we will not run these tests twice.

keys being pressed								Result	Note
A	B	C	D	E	F	G	H	✗	Original set
A	B	C	D					✓	Dividing into two subsets
				E	F	G	H	✓	
A	B							✓	Dividing into four subsets
		C	D					✓	
				E	F			✓	
						G	H	✓	
		C	D	E	F	G	H	✓	Using complements of subsets
A	B			E	F	G	H	✗	
A	B							✓	Working with $\{\{A, B\}, \{E, F\}, \{G, H\}\}$
				E	F			✓	
						G	H	✓	
				E	F	G	H	✓	Using complements of subsets
A	B					G	H	✓	
A	B			E	F			✗	
A	B							✓	Working with $\{\{A, B\}, \{E, F\}\}$
				E	F			✓	Splitting
A								✓	Working with $\{\{A\}, \{B\}, \{E\}, \{F\}\}$
	B							✓	
				E				✓	
					F			✓	
	B			E	F			✓	Using complements of subsets
A				E	F			✓	
A	B			E				✓	
A	B				F			✗	
A								✓	Working with $\{\{A\}, \{B\}, \{F\}\}$
	B							✓	
					F			✓	
	B				F			✓	
A					F			✓	
A	B							✓	
A	B				F			✗	DONE

Figure 2.2: DD in action: The minimization of the keyboard buttons results in $\{A, B, F\}$, which is the last failing subset. As it is show in this table, we also try to work with the complements of the potential solution in order to make the algorithm more efficient.

2.2.1 Formalizing Delta Debugging

The Delta Debugging technique has been defined in many publications as [11], [12], [13]. We will use most of the notation found in [6].

For this work, we are interested in the minimization of a failing test case of a system. If we assume that we have a system and a test case makes the system fail, we would like to minimize the test case by successive testing. This means that we want to obtain the minimal test case: if we remove any element from this test case, the failure disappears. We denote a failure in the test by the symbol ✗ and if the test case passes we use ✓.

In the rest of this section, we will use the term *circumstances* to refer to the data or external envirtonment affecting the execution of a specific software program. In particular, we are interested in those circumstances that could vary and cause a system to behave differently. We denote the set of possible configurations of circumstances by $\mathcal{C}$.

In order to identify the changes causing failures, we say that $r \in \mathcal{C}$ is a specific *program run*. Any program run r that fails is denoted as $r_{✗}$ and, analogously, if it passes we use $r_{✓}$. To talk about the modifications in the test cases, we introduce the following definition.

Definition 1. *A change δ is a mapping $\delta : \mathcal{C} \to \mathcal{C}$. When talking about changes between two runs $r_{✓}$ and $r_{✗}$, we say that δ is a relevant change if it makes $\delta(r_{✓}) = r_{✗}$.*

In general, we will represent a relevant change δ by a set of atomic changes

$\delta_1, \ldots, \delta_n$. These changes are problem specific. For instance, for the keyboard example above, the change between $\{A\}_{\checkmark}$ and $\{A, B, C\}_{\text{✗}}$ could be thought as $\delta = \delta_1 \circ \delta_2$ where δ_1 is the insertion of B to the set and and δ_2 is the insertion of C; the $\circ$ operation refers to the composition that groups changes.

Definition 2. *The function* $rtest : \mathcal{C} \to \{\text{✗}, \checkmark\}$ *states whether a run* $r \in \mathcal{C}$ *fails or not. In other words,* $rtest(r_{\text{✗}}) = \text{✗}$ *and* $rtest(r_{\checkmark}) = \checkmark$*. We also define* $c_{\checkmark}$ *as the empty set (*$c_{\checkmark} = \emptyset$*) and the set of all changes as* $c_{\text{✗}}$*. A test case is a subset* $c \subseteq c_{\text{✗}}$*.*

Now we can introduce the notion of test over $\mathcal{C}$:

Definition 3. *If we denote the power set of* $c_{\text{✗}}$ *as* $\wp(c_{\text{✗}})$*, the function* $test : \wp(c_{\text{✗}}) \to \{\text{✗}, \checkmark\}$ *is defined as* $test(c) = rtest((\delta_1, \ldots, \delta_n)(c_{\checkmark}))$ *where* $c = \{\delta_1, \ldots, \delta_n\}$ *is a test case. As a consequence of this definition,* $c_{\checkmark} = \checkmark$ *and* $c_{\text{✗}} = \text{✗}$*.*

Given a failing test set, we would like to find the minimal subset that also fails. Now we formalize what we mean by minimization.

Definition 4. *We say that a test case* $c \subseteq c_{\text{✗}}$ *is n-minimal if* $test(c') = \checkmark$ *for every* $c' \subseteq c$ *given that* $|c - c'| \leq n$*.*

Basically, the definition 4 states that a test case is n-minimal when any subset passes the test, given that the number of changes in the test case is less than or equal to n changes from the original test case. Delta Debugging was conceived to guarantee 1-minimality. This means that we are looking for a failing test case such that, when any single change is removed from it, the test case passes.

In order to obtain 1-minimality, we cannot just remove one element at a time. The way to do it is by means of binary search. Let us suppose that we are given

a failing set of changes $c_{\text{✗}}$. To get started, the original set $c_{\text{✗}}$ is divided into two subsets of similar size, which are called Δ_1 and Δ_2. Now, we proceed to test them. The possibilities are:

- Δ_1 fails. Then we continue reducing Δ_1 instead of $c_{\text{✗}}$, which is a smaller set.
- Δ_1 does not fail, but Δ_2 does. As before, we now reduce Δ_2 which is also smaller.
- Both tests pass. This case is what we call as *ignorance*.

The first two options consist on dividing the original set into a smaller set containing approximately one half of the elements. The search space is reduced by one half.

But, in case of ignorance, we should partition the set in order to get a failing subset. There are two ways to do this:

- If the subsets of $c_{\text{✗}}$ are large, then there is a greater chance that the test fails, but the progression of the algorithm gets smaller.
- If the subsets of $c_{\text{✗}}$ are small, the chances to find a test case that fails get smaller, but we increase the speed of progression of the algorithm.

Delta Debugging tries to take advantage of these two approximations by using both at the same time. When $c_{\text{✗}}$ is divided into n subsets, each one called Δ_i, the subsets are small so we get fast progression. But, simultaneously, we also make use of the complement of Δ_i: $\nabla_i = c_{\text{✗}} \setminus \Delta_i$. ∇_i is a larger test case having more chance to fail.

The problem is that we are increasing the number of test cases to test. If n is the number of subsets, then we should test $\Delta_1, \ldots, \Delta_n$ and their complements $\nabla_1, \ldots, \nabla_n$. We have three possibilities now:

- If Δ_i fails, we continue by reducing this subset. This means that we take Δ_i as the failing set and set $n = 2$ to continue. In practice, we are restarting the algorithm with a smaller subset as in a "divide and conquer" approximation.

- If ∇_i fails, we use ∇_i as the failing test. In this case, instead of setting $n = 2$, we set $n = n - 1$. The reason why we use $n - 1$ is because many of the large subsets of ∇_i have already been tested before. For instance, we have depicted the situation when $\nabla_5 = c_{\text{✗}} \setminus \Delta_5$ fails for $n = 8$ in figure 2.3. As we can see, $\nabla_5 = \bigcup_{i=1}^{4} \Delta_i \cup \bigcup_{i=6}^{8} \Delta_i$ and we have already tested $\Delta_1, \ldots, \Delta_8$. So, if we set $n = 2$ we will redo most of the work; instead, by setting $n = 7$, we continue working from the last partition ($\{\Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_6, \Delta_7, \Delta_8\}$) that we had used before.

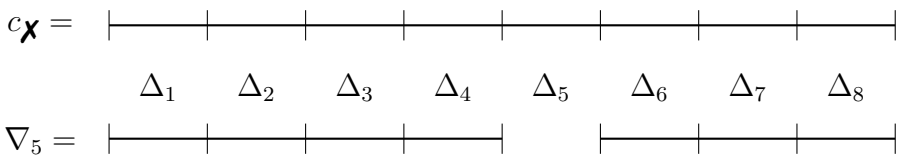

Figure 2.3: DD subsets partitioning

- If none of these subsets fail, we increase the granularity by setting $n = 2n$. This is done just when $2n > |c_{\text{✗}}|$; if the last condition does not hold, then the algorithm finishes.

Figure 2.4 briefly summarizes the algorithm. We continue using the same notation as in [6].

$c_{\text{✗}}$ is a test case satisfying $test\ (c_{\text{✗}}) = \text{✗}$ and $test\ (\emptyset) = \text{✓}$.

The Delta Debugging algorithm *ddmin*(c) is defined as follows:

$ddmin\ (c_{\text{✗}}) = ddmin_2(c_{\text{✗}}, 2)$

and

$$ddmin_2(c'_{\text{✗}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } test(\Delta_i) = \text{✗ for some } i \in \{1, \ldots, n\} \\ ddmin_2(\nabla_i, n-1) & \text{if } test(\nabla_i) = \text{✗ for some } i \in \{1, \ldots, n\} \\ ddmin_2(c'_{\text{✗}}, 2n) & \text{if } n < |c'_{\text{✗}}| \\ c'_{\text{✗}} & \text{o.w.} \end{cases}$$

Figure 2.4: Delta Debugging algorithm

2.3 topformflat

In the Delta Debugging source code implementation [14], Daniel S. Wilkerson and Scott McPeak include a tool called "topformflat", which aims to manage the hierarchical structure of source code.

Let us consider the problem of minimizing source code. For instance, we may consider the problem that a compiler is failing to compile a file, and we would like to get the minimal piece of code that produces exactly the same error. In order to complete this task, we could consider that a file is a sequence of lines and all the lines in the original source file represent the set of all changes. In this particular situation, the changes or lines of code strongly depend on each other. topformflat is a simple idea to handle the nesting structure of C/C++ code.

The *nesting depth* of a line of source code refers to the number of blocks that that line belongs to. This concept is language specific, this is why we are using C/C++ here and we will not formalize the definition. But, to get the intuition, let us consider the example in figure 2.5. We can see the nesting depth of each line in a source code file.

```
                                                                nesting depth
#include<stdio.h>                                                       0
int main (void) {                                                       0
  int i = 0,  a = 0;                                                    1
  printf("Please input an integer value: ");                            1
  scanf("%d", &a);                                                      1
  if (a > 100) {                                                        1
    printf("You entered: %d\n", a);                                     2
  } else {                                                              1
    printf("The multiples of 7 smaller than %d are: ", a);              2
    for (i = 7; i < 100; i+=7) {                                        2
      printf("%d ", i);                                                 3
    }                                                                   2
  }                                                                     1
  return 0;                                                             1
 }                                                                      0
```

Figure 2.5: Source code nesting depth

Given a desired nesting depth n and a source code file, topformflat creates a new source code such that every line whose depth is greater than n is concatenated to the previous line ignoring the newline characters between them. In other words, topformflat copies the input file into a new file such that the newline characters are omitted when the nested depth is greater than a nesting depth argument. We say that topformflat "flattens" the source code.

We can see an example of the result of topformflat in figure 2.6, when applied to the code given in the previous figure.

```
#include<stdio.h>
int main (void) { int i = 0,  a = 0; printf("Please input ...");    ...; return 0;
}
```

(a) topformflat level 0

```
#include<stdio.h>
int main (void) {
  int i = 0,  a = 0;
  printf("Please input an integer value: ");
  scanf("%d", &a);
  if (a > 100) { printf("You entered: %d\n", a);
  } else {  printf("The multiples ...", a); for (i = 7; i < 100; i+=7) { printf("%d ", i); } }
  return 0;
}
```

(b) topformflat level 1

```
#include<stdio.h>
int main (void) {
  int i = 0,  a = 0;
  printf("Please input an integer value: ");
  scanf("%d", &a);
  if (a > 100) {
    printf("You entered: %d\n", a);
  } else {
    printf("The multiples of 7 smaller than %d are: ", a);
    for (i = 7; i < 100; i+=7) { printf("%d ", i); }
  }
  return 0;
}
```

(c) topformflat level 2

Figure 2.6: Examples of topformflat for different nesting levels

topformflat can be integrated with Delta Debugging. The strategy consists of minimizing the topformflat result while increasing the nesting depths, starting from zero depth. We will describe this approach in detail in the next chapter.

2.4 Iterative Reduction Debugging

An alternative to Delta Debugging that could be useful in some domains is Iterative Reduction Debugging. From a starting failing ordered test case, this tech-

nique consists of removing all possible consecutive changes until a failing test case is found, in which case a new iteration is performed.

For this method to be defined, we have to make sure that the elements in the test case have a specific order, to make meaningful the notion of consecutiveness. More precisely, the set of all changes has to be a finite totally ordered set.

In this case, we say that $\nabla_i^j = c_{\text{✗}} \setminus \{e_i, \ldots, e_j\}$ where $c_{\text{✗}} = \{e_1, \ldots, e_m\}$ are the elements of $c_{\text{✗}}$ in order.

The algorithm is shown in figure 2.7.

$c_{\text{✗}}$ is an ordered test case satisfying $test\ (c_{\text{✗}}) = \text{✗}$ and $test\ (\emptyset) = \text{✓}$.

The Iterative Reduction Debugging algorithm *ird*(c) is defined as follows:

$ird\ (c_{\text{✗}}) = ird_2(c_{\text{✗}}, 1)$

and

$$ird_2(c'_{\text{✗}}, n) = \begin{cases} ird_2(\nabla_i^{i+n}, n) & \text{if } test(\nabla_i^{i+n}) = \text{✗ for some } i \in \{1, \ldots, m-n\} \\ ird_2(c'_{\text{✗}}, n+1) & \text{else if } n < m \\ c'_{\text{✗}} & \text{o.w.} \end{cases}$$

Figure 2.7: Iterative Reduction Debugging algorithm

Additionally, we can also define a special case of minimality for these test cases.

Definition 5. *If the set of all changes $c_{\text{✗}}$ has an associated order, a test case $c \subseteq c_{\text{✗}}$ is n-consec-minimal if $test(c') = \text{✓}$ for every $c' \subseteq c$ given that $|c - c'| \leq n$ and all the elements in $c - c'$ are consecutive.*

From this definition, it is trivial to prove that:

$$n\text{-minimality} \Rightarrow n\text{-consec-minimality} \Rightarrow 1\text{-minimality}$$

IRD guarantees n-consec-minimality for every number n: simply, from the original set, we remove n consecutive lines and try to generate the same error. If it woks, we continue, if it does not, we choose some other lines. Even though the method is very straightforward, in practice it works reasonably and its performance is similar to DD, but more general: it generates n-consec-minimal results.

Chapter 3: Source Code Reduction

Our main goal is to reduce source code files into smaller size files. The original source code is supposed to be a compilable program such that when we run a SCA tool on it, it generates a warning. This warning could be a false positive or a true positive. As we try to summarize false positives, we propose a technique to reduce the source code into valid snippets that, when the SCA tool is run on them, they generate exactly the same warning. In the rest of this work, we will refer to the resulting source code as the code snippets or simply snippets.

We begin this chapter by implementing the code reduction by means of Delta Debugging and then we continue improving the approach to reach the main goal.

3.1 Reduction by Delta Debugging

In the previous chapter, we introduced the DD technique. To apply it to code reduction, we have to define the test case $c_{\text{✗}}$, the *test* function and what the relevant changes are.

As we are given a source program $\mathcal{P}$ that compiles and, when used as input for a SCA tool, it generates a warning $\mathcal{W}$, we say that

$$test(c) = \begin{cases} \text{✗} & \text{if } c \text{ compiles and the SCA tool generates the warning } \mathcal{W} \\ \text{✓} & \text{if } c \text{ does not compile or it compiles but } \mathcal{W} \text{ is not generated} \end{cases}$$

where c is a source code file "reduced" from $\mathcal{P}$. Now we should define what we mean by a source code file obtained from a program, but this is done by means of a change. The changes in a program are the expansion of a trivial empty source code file by portions of $\mathcal{P}$.

A relevant change should be decomposed into a finite set of atomic changes. These changes, for this specific problem, could be the insertion of single characters into the source code, or adding tokens, or adding lines. In practice, using characters or tokens leads to the generation of spurious source code. We can see in figure 3.1 an example of this phenomenon. If we force the line 3 to be part of the reduced snippet, we could obtain misleading code, as in figure 3.1c.

```
1  int var1 = 0;
2  int var21 = 3;
3  printf("i = %d", var1);
```

(a) Original code to reduce, keeping line 3

```
1  int var1 = 0;
2  int var21 = 3;
3  printf("i = %d", var1);
```

(b) Removed characters

```
   int var1 = 3;

3  printf("i = %d", var1);
```

(c) Reduced spurious code

Figure 3.1: Code reduction with single caracters as atomic changes

The situation is similar if we consider tokens. This problem can be solved by taking lines of code as the atomic changes. Also, lines of code make the search space

smaller. Note that we expect the source code to reduce to be written guided by a common or friendly coding style.

Shown in figure 3.2c is an example of the process of code reduction by lines of code using the Delta Debugging approach. The original code generated a warning in a SCA tool stating that a variable was freed two times and the obtained snippet generates exactly the same warning.

```
1  int main (void) {
2        int * data = (int *) malloc(5 * sizeof(int));
3        int value = 10;
4        data[1] = value * 12 + 7;
5        free(data);
6        free(data);
7        return 0;
8  }
```

(a) Original source code

lines in c								Result	Next step description
1	2	3	4	5	6	7	8	✗	Increasing n (n = 2)
1	2	3	4					✓ (does not compile)	
				5	6	7	8	✓ (does not compile)	Increasing n (n = 4)
1	2							✓ (does not compile)	
		3	4					✓ (does not compile)	
				5	6			✓ (does not compile)	
						7	8	✓ (does not compile)	Trying with complements
		3	4	5	6	7	8	✓ (does not compile)	
1	2			5	6	7	8	✗	Reducing to ∇_2 (n = 3)
1	2							✓ (already tested)	
				5	6			✓ (already tested)	
						7	8	✓ (already tested)	Trying with complements
				5	6	7	8	✓ (does not compile)	
1	2					7	8	✓ (does not generate warning)	
1	2			5	6			✓	Increasing n (n = 6)
1								✓ (does not compile)	
	2							✓ (does not compile)	
				5				✓ (does not compile)	
					6			✓ (does not compile)	
						7		✓ (does not compile)	
							8	✓ (does not compile)	Trying with complements
	2			5	6	7	8	✓ (does not compile)	
1				5	6	7	8	✓ (does not compile)	
1	2				6	7	8	✓ (does not generate warning)	
1	2			5		7	8	✓ (does not generate warning)	
1	2			5	6		8	✓ (does not compile)	
1	2			5	6	7		✓ (does not compile)	
1	2			5	6	7	8	✗	

(b) DD algorithm based on the lines of the source code

```
1  int main (void) {
2     int * data = (int *) malloc(5 * sizeof(int));
5       free(data);
6       free(data);
7       return 0;
8  }
```

(c) Result of the reduction

Figure 3.2: Example of the process of source code reduction by means of DD

3.1.1 Analysis of Delta Debugging Source Code Reduction

Let us consider another similar source code file, the one in figure 3.3a. Again, a SCA tool will generate a warning stating that the variable "data" is freed twice. We would like to reduce the source code. Lines 3, 4 and 5 just initialize values in the array and they do not contribute to the warning itself. So, we expect that the resulting snippet consists of lines 1, 2, 6, 7, 8, 9. We can see in figure 3.3b that DD does not remove all these lines. In fact, DD just eliminates line 4.

John Regehr, in [15], analyzes Delta Debugging in detail and criticizes it. Delta Debugging is described as a "greedy search" algorithm in the sense that it removes a contiguous chunk of the input and when there are no options of chunks to be removed, it reduces the size of these chunks. The author states that DD could be suitable to perform minimization in specific contexts, like random testing. But, in some other domains, DD may find a local minimum before the test case is fully reduced. This may happen in structured test cases, like the ones we are dealing with. A computer program source code consists of several instructions forming a set of dependent structures. This is, in fact, the case where DD does not work as we expected.

If we consider the execution of DD again, we see that in the grey row of figure 3.3b, two lines that should be eliminated are ignored because the test passes. Here is where the structure of the source code is affecting the results. The code snippet obtained in this step does not compile because of the for loop syntax, but in a non-structured domain this test case would have failed. And in some way, DD is implicitly making the assumption that the *test* is monotone.

Definition 6. *A test function is monotone if given two sets of changes c and c', such that $c \subseteq c'$, then we have*

$$test(c) = \text{✗} \Rightarrow test(c') = \text{✗}$$

Equivalently, we can say that, given the same conditions, *test* is *monotone* if

$$\text{test}(c') = \checkmark \Rightarrow \text{test}(c) = \checkmark$$

The test set in the gray row, consisting of lines $\{1, 2, 3, 6, 7, 8, 9\}$, passes the test. Therefore, $\{1, 2, 6, 7, 8, 9\}$ should also pass and this test case is never created. Anyway, there is a really good reason not to create it: the complexity. If we try with every possible subset, the complexity becomes exponential.

More formally, the reason why DD is not finding the snippet we expected is because DD guarantees 1-minimality: if we remove any single line from the minimized result, the resulting test case passes. And that is, in fact, what is happening with the source code in figure 3.3b. We may need 2-minimality (or 2-conseq-minimality) for the lines 3 and 5 to be discarded.

```
1  int main (void) {
2        int * data = (int *) malloc(5 * sizeof(int));
3        for (int i = 0; i < 5; i++) {
4              data[i] = i;
5        }
6        free(data);
7        free(data);
8        return 0;
9  }
```

(a) Initial source code

lines in c									Result	Next step description
1	2	3	4	5	6	7	8	9	✗	Increasing n (n = 2)
1	2	3	4	5					✓ (does not compile)	
					6	7	8	9	✓ (does not compile)	Increasing n (n = 4)
1	2	3							✓ (does not compile)	
			4	5					✓ (does not compile)	
					6	7			✓ (does not compile)	
							8	9	✓ (does not compile)	Trying with complements
			4	5	6	7	8	9	✓ (does not compile)	
1	2	3			6	7	8	9	✓ (does not compile)	
1	2	3	4	5			8	9	✓ (does not generate warning)	
1	2	3	4	5	6	7			✓ (does not compile)	Increasing n (n = 8)
1	2								✓ (does not compile)	
		3							✓ (does not compile)	
			4						✓ (does not compile)	
				...					...	
								9	✓ (does not compile)	Trying with complements
		3	4	5	6	7	8	9	✓ (does not compile)	
1	2		4	5	6	7	8	9	✓ (does not compile)	
1	2	3		5	6	7	8	9	✗	Reducing to ∇_3 (n = 7)
1	2								✓ (already tested)	
		3							✓ (already tested)	
				...					...	
								9	✓ (already tested)	Trying with complements
		3		5	6	7	8	9	✓ (does not compile)	
1	2			5	6	7	8	9	✓ (does not compile)	
				...					...	
1	2	3		5	6	7	8		✓ (does not compile)	Increasing n (n = 8)
1									✓ (does not compile)	
	2								✓ (does not compile)	
		3							✓ (already tested)	
				...					...	
								9	✓ (already tested)	Trying with complements
	2	3		5	6	7	8	9	✓ (does not compile)	
1		3		5	6	7	8	9	✓ (does not compile)	
1	2			5	6	7	8	9	✓ (already tested)	
				...					...	
1	2	3		5	6	7	8		✓ (already tested)	
1	2	3		5	6	7	8	9	✗	

(b) DD algorithm based on the lines of the source code

```
1  int main (void) {
2        int * data = (int *) malloc(5 * sizeof(int));
3        for (int i = 0; i < 5; i++) {
5        }
6        free(data);
7        free(data);
8        return 0;
9  }
```

(c) Resulting code snippet

Figure 3.3: DD source code reduction resulting in extra lines

3.2 Reduction by Delta Debugging with topformflat

topformflat is a very simple idea that can be integrated with any code reduction algorithm. In this section we present it combined with DD. One of its main advantages is that it makes DD more efficient. It increases the processing speed significantly. Additionally, topformflat solves the problems of DD regarding the hierarchical structure of source code.

In the remaining of this work we will refer to this technique as Delta Debugging with topformflat integration (DDT). In figure 3.4, we detail the algorithm for DD.

$c_{\text{✗}}$ is a test case satisfying $test\ (c_{\text{✗}}) = \text{✗}$ and $test\ (\emptyset) = \text{✓}$.

The Reduction by DD with topformflat integration algorithm is then:

$maxlevel \leftarrow maxdepth(c_{\text{✗}})$
$level \leftarrow 0$
$c'_{\text{✗}} \leftarrow c_{\text{✗}}$
for $level \in \{0, \ldots, maxlevel\}$ **do**
  $c''_{\text{✗}} \leftarrow topformflat(c'_{\text{✗}})$
  $c'_{\text{✗}} \leftarrow ddmin(c''_{\text{✗}})$
end for

Figure 3.4: Delta Debugging algorithm with topformflat integration

As we mentioned before, in practice, DDT handles the source code hierarchical structure problems that DD per se has. We use the source code from the previous examples to show in figure 3.5 how this algorithm works. In particular, when the source code is flatten with depth level 2, as in figure 3.5a, we see how the entire "for loop" that is the structure that we want to remove, becomes a single line. DD will try to discard every single line of code in the last step of the algorithm.

Consequently, the loop dissappears from the result.

```
1  int main (void) {
2        int * data = (int *) malloc(5 * sizeof(int));
3        for (int i = 0; i < 5; i++) {
4                data[i] = i;
5        }
6        free(data);
7        free(data);
8        return 0;
9  }
```

(a) Initial source code

```
1  int main (void) {
2        int * data = (int *) malloc(5 * sizeof(int));
3        for (int i = 0; i < 5; i++) { data[i] = i; }
4        free(data);
5        free(data);
6        return 0;
7  }
```

(b) Source code after topformflat level 2

lines in c							Result	Next step description
1	2	3	4	5	6	7	✗	Increasing n (n = 2)
1	2	3	4				✓ (does not compile)	
				5	6	7	✓ (does not compile)	Increasing n (n = 4)
1	2						✓ (does not compile)	
		3	4				✓ (does not compile)	
				5	6		✓ (does not compile)	
						7	✓ (does not compile)	Trying with complements
		3	4	5	6	7	✓ (does not compile)	
1	2			5	6	7	✓ (does not compile)	
1	2	3	4			7	✓ (does not generate warning)	
1	2	3	4	5	6		✓ (does not compile)	Increasing n (n = 8)
1							✓ (does not compile)	
	2						✓ (does not compile)	
		...					...	
						7	✓ (does not compile)	Trying with complements
	2	3	4	5	6	7	✓ (does not compile)	
1	2		4	5	6	7	✗	
		...					...	
1	2		4	5	6	7	✗	

(c) DDT run on the flatten code

```
1  int main (void) {
2        int * data = (int *) malloc(5 * sizeof(int));
6        free(data);
7        free(data);
8        return 0;
9  }
```

(d) Reduction result

Figure 3.5: DDT source code reduction

3.3 Reduction by Iterative Reduction Debugging with topformflat

The Iterative Reduction Debugging algorithm requires an initial failing ordered test case. For source code reduction, we can define this set as the set of LOC with their corresponding line number. Their order is the natural one of the line numbers (so, for instance, the first line precedes the second line and so on). As stated before, this technique generates new test cases by deleting consecutive lines of code in blocks. When a test case fails, it continues the reduction from this last failing code.

IRD was defined to implicitly tackle the problem of figure 3.3: the hierarchical structure of code. By construction, this algorithm will find every single block and try to remove it.

It is reasonable then, to try to use the topformflat idea with IRD, we will call this technique as Iterative Reduction Debugging with topformflat integration (IRDT). The algorithm simply consists of calling the *ird* function instead of the *ddmin* function in figure 3.4. Through topformflat, IRD improves the performance efficiency notably.

3.4 Preserved Control Flow Reduction

All the source code reduction algorithms described in this chapter could add an extra feature: keeping the control flow of the original source code. The control flow or flow of control of a program makes reference to the order in which its statements are executed.

When reducing the source code, one may want to preserve the control flow of the original program. The main motivation to do this is given in the example of figure 3.6. This is a trivial example, but shows that the previous approaches may not work in practice. The difficulty that arises after the reduction process, is that a false positive is converted into a true positive. Line 5 in the code in figure 3.6a is death code. Even though it will never be executed, some SCA tools could report that the variable "data" is freed twice. If we reduce the source code into the one in figure 3.6b, we change the SCA warning completely: it was a FP and became a TP.

Remember that our goal is to reduce FPs to characterize them, in order to learn from them. If the FPs are turned into TPs, the characterization will be completely invalid.

```
1  int main (void) {
2        int * data = (int *) malloc(5 * sizeof(int));
3        free(data);
4        if (0) {
5                free(data);
6        }
7        return 0;
8  }
```

(a) Initial source code

```
1  int main (void) {
2        int * data = (int *) malloc(5 * sizeof(int));
3        free(data);
5        free(data);
7        return 0;
8  }
```

(b) Reduced source code generating the same warning

Figure 3.6: DD with topformflat source code reduction

To try to preserve the control flow of the original source code, we force some

specific lines to be part of the resulting snippet. These lines are called *forced lines*.

We assume that a warning was generated in a specific line of code n. To identify the lines that will be added to the forced lines, we first determine the blocks that n belongs to. The blocks that we consider are:

- control statements: like "if" conditions, "for" and "while" loops, "switch" statements.
- class definitions.
- method or function definitions.
- try/catch blocks.

The lines of code containing these statements, are added to the set of forced lines. Additionally, for these blocks we incorporate the lines corresponding to "break", "continue" and "return" statements.

Accordingly, adding these lines to the reduced source code tends to preserve the original control flow. This method can be an extension to DDT and IRDT. In the next section we show how this approach works in practice.

Chapter 4: CodeReducer tool

In this chapter we highlight the design of the tool, called "CodeReducer", and we also briefly describe its implementation.

4.1 Design and Implementation

The system is implemented in Python. The language was selected to quickly develop it while we focus on the techniques to reduce source code and on the evaluation of results.

The architecture of the system is described in figure 4.1.

In rough outline, the CodeReducer system consists of the following classes:

- Snippet Generator: the main class uses this class as an auxiliary to create the snippets from the original source code. To that end, its constructor takes the following parameters:
  - SCA tool to use to generate the warning
  - source code file generating a warning
  - compiler used to compile the file

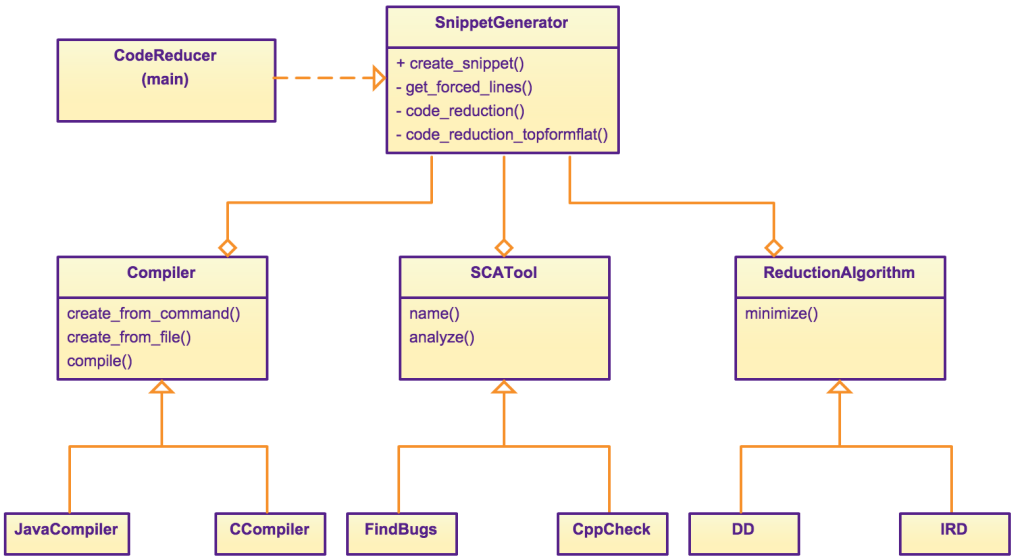

Figure 4.1: Class diagram of the system

  - line(s) where the warning was generated
  - warning message that the SCA tool reports

- SCATool: the CodeReducer supports different SCA tools. This class is intended to be extended by the different tools to perform the analysis of the original source code and the code that the CodeReducer generates in every step.

- Compiler: this class is an abstraction for the compilation process. As the CodeReducer supports multiple SCA tools, these tools find potential errors in different programming languages. In particular, we are using Java and C/C++ compilers.

- ReductionAlgorithm: The reduction algorithms described in the previous sec-

tion are inherited from this class.

The CodeReducer system was conceived to process the FPs obtained by the Static Code Analysis Tool Evaluator (SCATE) [16]. SCATE is a tool that provides a methodology to evaluate the quality of SCA tools based on test suites. In particular, SCATE generates reports from test suites created by the National Security Agency's (NSA) Center for Assured Software (CAS) to use in testing static analysis tools. SCATE informs what the FPs are for a specific tool and where these are located in the source code.

Besides the SCA tool warnings, the CodeReducer also needs to compile files. The tool has a parameter to define the compiler to be used and its options. Another possibility is to pass a json file similar to the compile_commands.json file produced by CMake to specify how files are compiled.

Chapter 5: Experiments

In this section we aim to evaluate empirically the system implemented. The reduction algorithms presented transform an original source code file into, by construction, a correct (compilable) source code snippet. But we are modifying the source code and we wonder if these changes will significantly impact on the original code semantics. We would like the snippets to remain as "similar" to the input as possible.

In this chapter we describe how we conducted the experiments and then we analyze the obtained results.

As pointed in section 3.1.1, reducing source code by simply using Delta Debugging generates code snippets that are 1-minimal. But we would like to remove as much code as possible from the snippets. Therefore, we will focus on the source code reduction by means of DD with topformflat integration (DDT), IRD with topformflat integration (IRDT) and the possibility to preserve the control flow in the snippet.

In general, we would like to quantify the code reduction in terms of LOCs and to know if the code snippets generated are meaningful to represent the original FPs.

To be more specific, we want to know how different the snippets generated

by DDT and IRDT are. These techniques have different minimization algorithms, so we expect them to produce different results. But the fact that topformflat is gradually applied to the source code could lead to similar code snippets.

Additionally, the DDT and IRDT techniques can be extended to preserve the control flow. We would like to know if there is any motivation in practice to use this extension and how results differ.

To entirely describe the empirical experiments, first we detail the testing data used and then we report the results.

5.1 Tests data

To reduce source code we have to determine the source code under test, the code that is going to be analyzed. We choose two approaches. The first one is to use open source projects. These are more realistic because the results will represent what a developer would obtain in practice. But the main problem is that we have to manually identify the false positives. The other alternative is to use test suites designed to evaluate SCA tools. The advantage of these test suites is that one could easily identify true positives and false positives. But these test suites may contain more artificial code.

We report our evaluation on the following open source applications/libraries, downloaded from GitHub and sourceforge:

- JUnit, version 4.11
- Java-design-patterns, version 1.5.0

- KDG Commons, version 1.0.5
- KAnalyze, version 0.9.7
- PMD, version 5.3.0

The National Institute of Standards and Technology (NIST) supports a project called the Software Assurance Reference Dataset (SARD). Acording to NIST [17], the purpose of the SARD is to provide users, researchers and software developers with a set of known security flaws. Consequently, the SARD has developed several test suites to evaluate SCA tools. In particular, we have used two version of Juliet as the test suites [1].

Juliet Test Suite for C/C++ version 1.1

Juliet 1.1 [18] is a set of test cases targeting difference flaws. Each of the test cases contains flawed and non-flawed functions that could be used to determine false positives and false negatives.

The flaw types for the test cases were selected taking into account the flaw types used in previous test suites, the team's experience in Software Assurance, the SCA tools vendors information about the types of flaws they identify and the weakness information in the Common Weakness Enumeration (CWE). CWE is a software project that created a catalog of the common software weaknesses and vulnerabilities.

[1] IARPA STONESOUP Phase 3 was also used but no warnings were generated for the SCA tool under test.

The test suite has some limitations due to the nature of the test cases. Many of them could be considered extremely artificial, meaning that they do not occur in practice. Also, the frequency of the flaws and non-flaws may not be the real one in natural software.

Juliet Test Suite for C/C++ version 1.2

SARD launched the version 1.2 of the Juliet test suite [19]. This version is an improvement from the previous release. Some of the changes include additional CWEs, removal of certain test cases from some CWEs and the number of flaw types being changed. Furthermore, dead code was removed from diverse control flow variants.

5.2 SCA tools

We have run experiments with two open source SCA tools. One for C/C++ and one for Java source code programs.

CppCheck

Cppcheck [20] is a SCA tool for C/C++ source code. This tool does not detect syntax error and it aims to detect the types of bugs that compilers do not detect, having zero false positives. It supports non-standard code including many compiler extensions and it works on any platform.

The documentation states that CppCheck rarely reports false positives and

there are many bugs that it does not detect. We are using CppCheck 1.68.

FindBugs

One of the most widely used SCA tool for Java source code is FindBugs [21]. FindBugs is also available under an open source license. The version used is 3.0.1.

The technique that it implements to find bugs in the software is based on bug patterns. Bug patterns are code idioms that are highly probable to be real vulnerabilities. The tool scans the Java bytecode of the program under analysis to find these patterns. The current version has 424 bug patterns organized into nine categories.

5.3 Results and Analysis

It is important to remark that we are not worried about the performance of the system. Consequently, the execution time will not be reported. We just mention that, depending on the SCA tool and the testing data, the time to reduce a source code file varies from a few seconds to about 20 minutes.

First we compare DDT and IRDT in terms of reduction of code or LOC of the resulting code snippet. We exectuted CodeReducer configured to run **DDT** and **IRDT** with **CppCheck** on a subset of **Juliet 1.1**. The subset was obtained randomly from all the test cases. We did work on a subset because we manually validated the false positives. Table 5.2 presents the results we got in terms of the size of the code (LOC).

Reduction	SCA Tool	Test Data	Source Code		Code Snippet	
			Mean (LOC)	SD (LOC)	Mean (LOC)	SD (LOC)
DDT	CppCheck	Juliet 1.1 (subset)	198.6	31.0	12.6	1.5
IRDT	CppCheck	Juliet 1.1 (subset)	198.6	31.0	11.2	1.3

Table 5.1: Comparison of DDT and IRDT

The code reduction is very significant in both cases, and the size of the code snippets is almost identical. We manually checked the snippets and found that DDT generates code with the same or more lines than IRDT. In figure 5.1, we show an example of the results for both approaches. In particular, the extra lines are due to the presence of C preprocessor conditionals that DDT fails to remove. As the conditional compilation statement "#ifdef" does not have a nested structure, topformflat is not capable to flatten them and, therefore, the DDT result contains those lines. On the other hand, IRDT will remove the last four lines because they are consecutive lines. Accordingly, we will keep our analysis on IRDT in the remaining of this section.

```
#include "std_testcase.h"
#ifndef OMITBAD
static void bad_sink(long long * data)
{
    data = (long long *)malloc(100*sizeof(long long));
}
#endif
#ifdef INCLUDEMAIN
#ifndef OMITGOOD
#endif
#endif
```

(a) DDT code snippet

```
#include "std_testcase.h"
#ifndef OMITBAD
static void bad_sink(long long * data)
{
    data = (long long *)malloc(100*sizeof(long long));
}
#endif
```

(b) IRDT code snippet

Figure 5.1: Comparison of code snippets generated with DDT and IRDT

We proceeded to manually inspect the code snippets generated by IRDT, to analyze them. In many cases the results were reasonably, the code generating a FP was changed to a little snippet causing the same FP. But in some other cases, we found two difficulties. We are going to refer to them as *snippets weaknesses*. The first one involves the problems mentioned in section 3.4, were a FP becomes a TP in the snippet. The other one is due to critical changes to the code semantics. An example of this situation is presented in figure 5.2, where two functions are merged to create a recursive one.

```
static int is_delete_array = 0;
static void free_helper(int * data) {
    if(is_delete_array)
    {
        /* INCIDENTAL: CWE 561 Dead Code, the code below will never run */
        /* POTENTIAL FLAW: Deallocate memory using delete [] - the source memory
         * allocation function may require a call to free() to deallocate it */
        delete [] data;
    }
    else
    {
        free(data);
    }
}
static void alloc_and_free() {
    int * data = NULL;
    /* POTENTIAL FLAW: Allocate memory with a function that requires free() */
    data = (int *)realloc(data, 100*sizeof(int));
    is_delete_array = 0; /* false */
    free_helper(data);
}
```

(a) Initial source code

```
static void free_helper(int * data) {
    {
        delete [] data;
    }
    data = NULL;
    data = (int *)realloc(data, 100*sizeof(int));
    free_helper(data);
}
```

(b) Source code reduced by IRDT

Figure 5.2: Severe semantic changes to the source code

In order to quantify these weaknesses, we count how often they occur in the experiments. The table 5.2 shows the results. For Juliet 1.1, we analyzed 135 source files producing FPs. Reducing them by IRDT resulted in 126 weaknesses. In 25 cases the problems were due to semantic changes as the ones described before and 101 times the original FP was converted into a TP in the snippet. As we want to characterize FPs, we would like the snippet to cause a FP too. For Juliet 1.2, we analyzed 37 FPs and only a few of them generated significant semantic changes or were converted into TPs. This may be because Juliet 1.1 has many test cases

consisting in death code that was changed in the 1.2 version.

Reduction	SCA Tool	Test Data	Files analyzed (FP)	Code Snippet Weakness	
				Semantic Change	Converted into TP
IRDT	CppCheck	Juliet 1.1	135	25	101
IRDT	CppCheck	Juliet 1.2	37	2	3

Table 5.2: Analysis of IRDT results

In chapter 3, we proposed an extension for DDT and IRDT to preserve the control flow of the original code. We would like to determine if this approach decreases the number of snippet weaknesses generated by IRDT. We test it on Juliet 1.1 and 1.2 and the results are displayed in table 5.3.

Reduction	SCA Tool	Test Data	Files analyzed (FP)	Code Snippet Weakness	
				Semantic Change	Converted into TP
Control Flow IRDT	CppCheck	Juliet 1.1	135	0	4
Control Flow IRDT	CppCheck	Juliet 1.2	37	0	0

Table 5.3: Analysis of Control Flow preserved IRDT results

We can see that the number of weaknesses in the snippets generated by IRDT have notably decreased when the control flow is preserved. For Juliet 1.1, 93% of the snippets were incorrectly generated, and with this approach just 3% of them could be considered wrong. For Juliet 1.2 the percentage is reduced from 14% to 0%. The four cases in which the FP has been changed into a TP are situations as in figure 5.3. We can see that, even thought we tried to force the control flow to remain as it was, the condition has changed forcing the control flow to change.

```
int main (void)
{
    int * data = NULL;
    data = new int [8];
    bool condition = true;
    condition = false;
    if (condition) {
        free(data);          # this is dead code, but a SCA tool generates a warning
    }
    delete [] data;
}
```

(a) Initial source code

```
int main (void)
{
    int * data = NULL;
    data = new int [8];
    bool condition = true;
    if (condition) {
        free(data);          # the SCA tool generates a warning on this line
    }
    delete [] data;
}
```

(b) Source code reduced

Figure 5.3: Persisting problem in Preserved Control Flow IRDT reduction, the original FP is turned into a TP

In terms of the size of the code snippet, preserving the control flow forces several lines of code to be part of the snippet. As a result, the number of LOCs of the resulting code will increase. In table 5.4 the LOCs are reported. If we compare these results with the ones in table 5.2, we can see that this version increases the length of the snippet between 50% and 100%, but the number of LOC remains low if we compare them with the original source code.

Reduction	**SCA Tool**	**Test Data**	**Source Code**		**Code Snippet**	
			Mean (LOC)	**SD (LOC)**	**Mean (LOC)**	**SD (LOC)**
Control Flow IRDT	CppCheck	Juliet 1.1 (subset)	125.3	22.6	19.4	3.5
Control Flow IRDT	CppCheck	Juliet 1.2 (subset)	112.6	22.5	15.0	3.3

Table 5.4: Control Flow preserved IRDT snippets size

To validate these results, we test the preserved control flow IRDT reduction

using FindBugs as the SCA tool. The next table shows the results we got for the different open source projects that we used.

Test Data	**Source Code**			**Code Snippet**		
	# FP	**Mean (LOC)**	**SD (LOC)**	**# Weaknesses**	**Mean (LOC)**	**SD (LOC)**
JUnit	2	266	61	0	15.5	0.5
Java Dessign Patterns	7	24.7	9.1	0	9.4	3.5
KDG Commons	1	60	—	0	18	—
PMD	1	183	—	1	24	—
KAnalyze	4	200.3	168.5	0	19.5	5.4

Table 5.5: Control Flow preserved IRDT snippets with FindBugs

We can see that the size of the code snippets in terms of LOC remains low and, for these test sets, there is just one snippet weakness: the system changed the nature of one warning, it was a FP and became a TP. However, just one FP in a set of 15 FPs was changed to a TP, which corresponds to 6% of them. We believe that, based on these numbers, the code snippets generated represent a suitable summarization of the false positives.

Chapter 6: Related Work

Since the Delta Debugging technique was introduced [11], many publications focused on automated test case reduction. But, a few papers concentrate on minimizing failure-inducing programs. Hierarchical Delta Debugging (HDD) [22] proposes an interesting procedure to use DD in hierarchical data. HDD reduces a tree-structured input by providing primitives for tree pruning. Every time a node of the tree is eliminated, its children are also removed, which makes it an efficient algorithm. A different way to do HDD would be to use another abstract representation of the source code, besides the ASTs. However, we found these approaches to be comparable to topformflat, this is why we did not apply it. An open research question is how similar these techniques are.

Regarding false positives characterization, as far as this author knows, the only publication that implements a learning system to predict SCA false positives is [4]. This system uses many factors obtained from source code metadata like file age, revision number, faults in previous releases. But it does not take into consideration the essence of the source code, that is the code by itself. We believe that, in order to obtain even better results, the source code must be analyzed. Also, we are suggesting an approach that, in a future, could help to classify the SCA tools warnings on any code, even though there is no development history such as its revision number.

Chapter 7: Conclusion

This thesis has presented different techniques with the aim of summarizing source code when a false positive is generated by a SCA tool. We introduced different methods that made the resulting code snippet similar to the original code, but we drastically reduced their size. These snippets are a simplified version of the same false positive.

The results suggest that one of the approaches makes the summarization feasible. Even though it may fail to keep the false positives in the obtained code snippets, the number of cases in which this occurs is found to be relatively small.

We believe that the technique is promising in terms of future work. As we have a simplified representation of the source code causing false positives, the next step is to characterize them. First, the minimized code snippet should be represented in a language independent manner, in order to compare code structure across different systems. Then, the abstract models of the false positives should form a catalog, to help in the learning process. Initially the learning activity can be based on different features obtained from the code snippets, the SCA tool reports, the type of error and any other contextual information of the environment where the warning was generated.

Bibliography

[1] Vinicius Rafael Lobo de Mendonca, Cassio Leonardo Rodrigues, Fabrizzio Alphonsus A de Soares, and Auri Marcelo Rizzo Vincenzi. Static analysis techniques and tools: A systematic mapping study. In *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, pages 72–78, 2013.

[2] Andy Chou. False positives over time: A problem in deploying static analysis tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[3] Nathaniel Ayewah and William Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5. ACM, 2008.

[4] Joseph R Ruthruff, John Penix, J David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, pages 341–350. ACM, 2008.

[5] Tim Boland and Paul E Black. Juliet 1.1 c/c++ and java test suite. *Computer*, (10):88–90, 2012.

[6] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.

[7] R. Dean Hartwick. Test planning. In *Proceedings of the June 13-16, 1977, National Computer Conference*, AFIPS '77, pages 285–294, New York, NY, USA, 1977. ACM.

[8] Wolfgang Wögerer. A survey of static program analysis techniques. *Vienna University of Technology*, 2005.

[9] Frank Elberzhager, Jürgen Münch, and Vi Tran Ngoc Nha. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Inf. Softw. Technol.*, 54(1):1–15, January 2012.

[10] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[11] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Software EngineeringESEC/FSE99*, pages 253–267. Springer, 1999.

[12] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. *arXiv preprint cs/0012009*, 2000.

[13] Andreas Zeller. Automated debugging: Are we close? *Computer*, 34(11):26–31, 2001.

[14] Daniel S. Wilkerson and Scott McPeak. Delta debugging implementation. `http://delta.stage.tigris.org/`. Accessed: 2015-08-02.

[15] John Regehr. Generalizing and criticizing delta debugging. `http://blog.regehr.org/archives/527`. Accessed: 2015-08-02.

[16] 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.

[17] National Institute of Standards and Technology. The nist software assurance reference dataset project. `http://samate.nist.gov/SARD/`. Accessed: 2015-08-02.

[18] Center for Assured Software United States National Security Agency. Juliet test suite v1.1 for c/c++. [Online]. Available: `http://samate.nist.gov/SARD/resources/Juliet_Test_Suite_v1.1_for_C_Cpp_-_UserGuide.pdf`, December 2011.

[19] Center for Assured Software United States National Security Agency. Juliet test suite v1.2 for c/c++. [Online]. Available: `http://samate.nist.gov/SARD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_UserGuide.pdf`, May 2013.

[20] Daniel Marjamäki. Cppcheck - a tool for static c/c++ code analysis. `http://cppcheck.sourceforge.net/manual.pdf`. Accessed: 2015-08-02.

[21] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[22] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM, 2006.