Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

**Callum Connolly**

May 26, 2020

---

# Investigating and Improving Automatic Software Repair of Security Vulnerabilities by Automatically Generating Test Suites

---

*Project supervisor*:
Dr Julian Rathke

*Second examiner*:
Dr Igor Golosnoy

A project progress report submitted for the award of
BSc Computer Science

# Abstract

Security defects are some of the most critical code errors that exist today. From remote code execution to data breach, vulnerabilities can cause untold damage and destruction. As such there is a great strive towards the automation of securing code. Automatic program repair, a relatively new area of research has the potential to fix code errors and relies solely on passing and failing test cases that come with software. Program repair currently is limited to using hand-made test suites - a process which takes a great deal of time and leaves plenty of room for human error.

In an effort to solve these problems we have developed two systems: SECdefects and SSG. SECdefects is a suite of security defects which provides a framework to assess APR tools. SSG is a proof-of-concept test suite generator capable of creating hundreds of test cases on-the-fly. Utilizing these programs, we have found that general program repair tools are unfortunately insufficient for repairing the majority of security defects. However, we have found test generation to be an effective method at providing high coverage test suites which are largely as effective as manual tests. Test generation also shows many promising avenues for future work.

## Acknowledgments

I would like to thank my supervisor Dr. Julian Rathke for his supervision and considerable patience during the course of my BSc Individual Project. I would also like to thank my second examiner Dr. Igor Golosnoy for his time and contribution.

I would like to thank my significant other Rebeccca Payne for her constructive criticism of the manuscript and support.

## Statement of Originality

I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students. I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme. I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes. I have acknowledged all sources, and identified any content taken from elsewhere. I did all the work myself, or with my allocated group, and have not helped anyone else. The material in the report is genuine, and I have included all my data/code/designs. I have not submitted any part of this work for another assessment. My work did not involve human participants, their cell or data, or animals

# Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Background and Motivation

Debugging software is often the bottleneck of software development, and can be both a time-consuming and budget-consuming process. This process can take up to as much as 50% of the budget, with software defects becoming more and more costly the later they are fixed.[1] Traditionally, the process of locating, fixing, and validating these faults is done either manually, or with limited tool support. Therefore, there has been a great drive to improve this process.

Automatic Program Repair (APR) is a relatively new approach which aims to automate the repair of software defects. Since there is such potential to significantly reduce the cost of debugging, it has received increasing attention in literature [2]. An analysis of papers featured on program-repair.org, a community driven website facilitating APR research shows an increasing number of publications year upon year as illustrated in Figure 1. However, as this is a developing field there are still a great many obstacles it faces.

Security defects, a subset of software defects which APR has the potential to fix, are some of the most critical code errors that exist today and can lead to significant problems. Security defects can be defined by 2 properties:

- Exploitation of these defects can lead to a security vulnerability in the software.
- These defects do not necessarily interrupt normal execution of the program.

The damage from security vulnerabilities being can lead to any number of issues, including remote code execution, malware infection, and data breach. The loss of data in a breach can cause devastating damage to an organisation's reputation and cause further impact on those whose records were leaked. In the first half of 2018, 3.3 billion records were leaked in breaches.[4]

It is important to note that not all defects will follow this pattern and those that do not will present their own challenges. However, when dealing with this area there are a number of assumptions we must make: Security defects need not be identified nor exploited to qualify as such, there are more unknown defects than known ones, and vulnerabilities in all programs must be assumed to be constantly under search and attack from various threat actors, especially those programs with high-value assets. The importance of finding security defects in programs before they are released into a production environment is not to be understated.

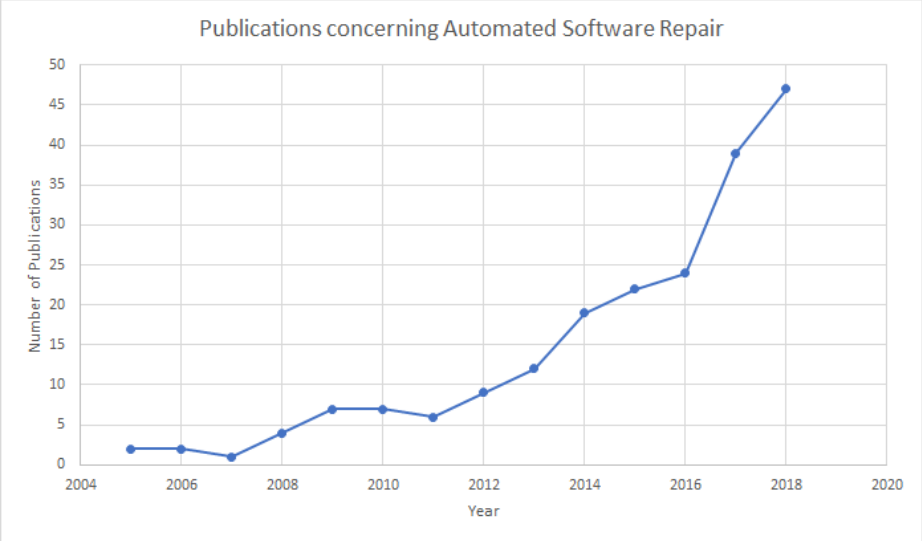


Figure 1: Publications featured on program-repair.org [3]

Although security defects have been covered extensively with static analysis, which will be examined in more depth later on, they have not been covered a great deal with dynamic analysis. Only a few approaches exist which often only target only a single vulnerability. [5][6][7] This is because each vulnerability requires a different fix and security defects are often more complex in nature than software defects. As such, there is a great deal of potential research to conduct in this area. It should be noted that both static and dynamic analysis of programs exists only in one small section when it comes to preventing security defects, [8] and should be used in conjunction with other methods.

Current APR techniques produce new programs based off of passing and failing test cases by editing sections of the source code itself to produce new programs. There are validated against the tests to ensure they function correctly. Test suites have been shown to be the dominant factor not only in producing better fixes, but, but also quicker fixing.[9] That being said, whilst running more efficient test suites reduces overheads in APR tools, the move from a manual to automatic debugging process is a much more meaningful improvement.

## 1.2 Project Goals

The aim of this project is to answer four main research questions, which have been detailed below:

- **RQ1:** How effective are current APR techniques regarding security defects?

  For this we will need to examine and investigate the current literature thoroughly and perform an assessment of APR tools on a bug-set of security defects.

- **RQ2:** How feasible is automatic test suite generation for improving program repair?

  We will examine how to go about automatically generating tests. Examining the manual vs generated tests will give an idea of the usefulness of this technique.

- **RQ3:** What are the contributions we can make towards improving program repair for security defects?

  As we go through, we will examine where the gaps in the field currently lie and see what we can contribute and learn.

## 1.3 Summary

In this section, we have examined the background to the problem and have seen that this is a new research area with plenty of room for improvement.

# 2 Literature Review

Developing solutions for security vulnerabilities by creating patches is often the bottleneck of the development process[10] with software maintenance taking up to 90% of the budget.[1] Furthermore, if these defects go unfixed, they can seriously affect the users of the system in a data breach and cause serious harm to a companies or organizations reputation through loss of user confidence. As such there is a great deal of interest in APR which allows programs to be fixed in-house while a vastly reduced team of debuggers only review the process. Already, there are a number of research papers that exist purely to evaluate APR tools against benchmarks of common software bugs, which show promising results.[11][12][13] Unfortunately, as this is a fairly new field, none so far exist which have examined the effectiveness of Automatic Software Repair techniques for fixing security defects.

The process of automatically repairing programs, illustrated in Figure 2, is iterative, with adjustments to the program made at the granularity of the Abstract Syntax Tree (AST). The process can be split into three steps:

1. **Localization**
   A set of locations for potential fixes are deduced from the passing and failing test cases of the program. These locations are assigned a suspicion based on the number of times they are executed in failing test cases.

2. **Fixing**
   A patch for the program is generated by applying certain types of change operators to these locations in order to produce a new program. More suspicious statements are prioritised for change.

3. **Verification**
   The candidate program is validated against the test suite to check that not only is the program syntactically correct, but has actually repaired the defect.

A quick word on localization; whilst localization it is a very important step in the process of software repair it is outside the scope of this project. This project focuses on the fixing and validation of the programs, as this is where the security specific nature of the problem lies. Additionally, there are a number of effective methods to go about localization such as spectrum-based, which is heavily utilized by a number of different APR tools.[14] That is not to say, however, that there is room for improvement.

Since an influential paper[15] first successfully exploited search based techniques to develop fixes, there have been a number of different tools which have been developed. Two techniques in particular have shown great promise and results: generate-and-validate techniques and semantics-based techniques.


Figure 2: Software repair process taken from[2] ©2017 IEEE

## 2.1 Current Techniques

### 2.1.1 Generate-and-validate

These techniques approximate the problem of generating fixes by defining a search space and then generating potential solutions. A potential solution is defined by a change to the source code of the original program at the granularity of the AST using some form of change operator.

Exploring this entire search space is generally intractable - running every possible fix at every possible location and then validating this synthesised program will take far too long. Genetic programming, a stochastic method which evolves solutions to solve a problem, has become a promising heuristic for examining this search space. It is widely used in various APR tools [16][17] and is the main approach for generating candidate solutions. This approach is illustrated further in Figure 3.

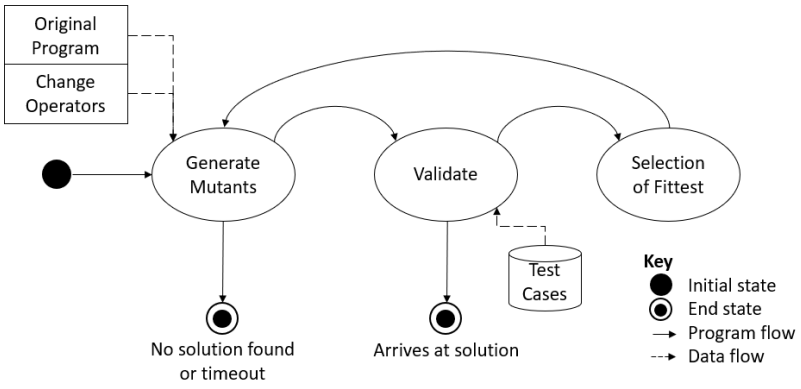


Figure 3: Evolution based generate-and-validate techniques (adapted from[2])

The change operator used will be different depending on the tool, and context of the problem. These can be grouped into three main types.

#### Atomic

Atomic change operators work for the simple program repair and only make changes to software by altering predicates or swapping statements. This technique lends itself as the most general approach to fixing software defects in programs[18], where an example set of this kind of change operator would be $<$, $>$, $<=$, $>=$, $=$. GenProg[16] is a well-known example of such an approach. Another program is Kali[12] which somewhat unconventionally solely uses functionality deletion as the change operator.

#### Example

According to the Plastic Surgery Hypothesis[19, p. 306] changes in "the content of new code can often be assembled out of fragments of code that already exist in the code base" and was formalized and validated in the same paper. Example based operators rely on this idea - that a defect has a correctly written implementation elsewhere in the program. This can be expanded, such that the code base can be derived from the program e.g. Cardumen,[20] from a large collection of similar programs,[21] or from a database of human written patches i.e. PAR.[22] Example based approaches use these gathered templates to coherently affect the program in potentially more than one location and have shown auspicious results.[20]

#### Template

These are predefined, often hand-written, sections of code used for solving very fault specific classes of problems i.e. ARC, which produces fixes for concurrency defects with non-trivial templates

such as locking and synchronization methods.[23] A proof of concept paper from a student at the University of Southampton has demonstrated the effectiveness of using predefined templates for matching specifically for security defects.[24]

### 2.1.2 Semantics-based

Semantics-based techniques follow a different approach; these produce an encoding of the problem that needs fixing, and then attempt to produce a solution to this problem by fixing the encoded problem. A solution to the encoded problem is guaranteed to solve the defect[25]. This approach can be similarly split into 3 steps:

1. **Behavioural Analysis**
   Exploiting information from test cases and the original program, analysing and extracting semantic behaviour about the intended functionality of the program.

2. **Problem Generation**
   Exploiting information gathered in the analysis stage to generate a formal representation of the program and potential changes, each change being an actual change in the code of the program itself.

3. **Fix Generation**
   Iteratively consider different locations for the fix, identifying the code change required in the problem generation step and applying these where possible to produce a potential solution.

Whilst semantics-driven methods require the most time and effort to get working they can be very effective in certain situations – this is because they often attempt to fix specific types of faults rather than being general purpose repair algorithms. It is much easier to come up with a formal representation of the problem when you know the specific characteristic of the program that needs fixing, however there is not a great deal research done in this area.

### 2.1.3 Novel Methods

As this field matures, naturally more novel techniques have begun to emerge.

#### Property-based

This is an especially novel approach extending semantics-based techniques and has been specifically designed with security defects in mind, solving a great many problems that exist with other techniques. By extending symbolic analysis, and in addition detecting and enforcing sets of human-written, vulnerability-specific safety properties, Senx[26] can generate a patch which effectively guarantees correct fixing of the code without side effects. Whilst it is an extremely resource expensive process it has already fixed 32 out of 42 real-world security defects.

#### Machine Learning

R2Fix [21] uses machine learning techniques to identify the real-world bug reports similar to the fault report. The identified bug reports are analysed and automatically paired with the associated predefined templates. These templates are then applied to the code with candidate solutions validated for correctness.

A recent PhD thesis from the University of Toronto examines security vulnerability detection using machine learning algorithms at both source code and bytecode level, with substantial results on synthetic bugs, but performs poorly on production code.[27]

Whilst it is difficult to procure large sets of meaningful data in this field, either for faulty programs or test suites, machine learning has demonstrated promise. Although, it can potentially be a little unclear what these algorithms actually learn, if anything.

## 2.2 Security Defects

There have been a few fixes for very fault specific security defects already, however none of these cover a wide range of faults. BovInspector[5] suggests fixes for buffer overflow defects found using symbolic analysis; CDRep[7] targets android cryptography misuse; SemRep[28] focuses on string sanitization and validation using forward and backward string analysis, which covers a larger range of potential defects than any previous security fault specific tool.

We can see that little to no literature exists on the effectiveness of general APR tools on a range of security defects. APR techniques mentioned previously will be theoretically assessed for effectiveness.

Atomic Based change operators generally do not have the complexity required to fix security defects - this is because security defects are quite complex in nature and often require more than one change at the level of the AST to fix them. Example based approaches could show some promise towards repairing security defects, with examples taken from a database or from the program itself. Template based methods are also an effective, with human-written templates being written by experts allowing for more complex sets of defects to fixed, ideal for security. As we have seen, this has already been applied with good results.[24]

Ideally, property-based APR would be would be examined in more detail, however this is far beyond the scope of the project both in terms of ability and time.

### 2.2.1 Static and Dynamic Analysis

Static analysis for program repair does not involve executing the program or test cases, whilst dynamic repair does, enabling it to find more subtle software and security defects[29]. A research effort by NASA concluded static analysis to be an inadequate guarantee of software assurance[30]. Nevertheless, static analysis methods have matured. Now programs exist which check code signatures, insecure API calls and insecure dependencies and, fortunately, have become very well integrated with the software development workflow[31]. In conjunction with other methods, both static analysis and dynamic repair have their place in fixing security defects and the combination of utilizing static and dynamic methods together is an important area of research.

## 2.3 Test Suites

Test suites, or validation suites, show that a program behaves to some specification. They are fundamental to program repair, however have not received a great deal of attention in literature. Furthermore, most experimentation that has been conducted only provides empirical evidence towards usefulness. Test suites have been shown to be the dominant cost factor in generate-and-validate techniques[2]. Traditional test suite metrics proposed for software testing e.g. branch coverage, code coverage, have been shown to be an effective measure of the quality and acceptability of a patch produced by a APR tool.[32] Additionally, there is some evidence to indicate that a good amount of failing test cases may be useful.[33] There still exists a great deal of research in this area and test suites in general are not designed for program repair[2] which is something that could potentially be further researched in time. So far there exists no literature covering the use of automatic test suite generation for security vulnerabilities, and the potential impact on program repair.

## 2.4 Summary

In this section we have assessed the viability of using APR tools for security defects, examined static and dynamic analysis, as well as the utilization and research on test suites. We can conclude that there is plenty to contribute in this area and now move forward with analysing some of the the ideas found.

# 3 Analysis

## 3.1 Current APR Techniques for Security Defects

Java is one of the most popular programming languages and runs on a number of different platforms. Therefore, the number of bugs to potentially fix, as well as the number of APR tools to potentially use, makes it an ideal language for research. For these reasons, and in keeping with initial practical audit skills (see Appendix A) Java was selected as the language of choice for this project.

To begin evaluation of the tools, we will have to start by finding a reasonable selection of APR tools and spend time understanding how these work. Afterwards, we will look towards finding a suitable set of bugs, ideally with buggy source code, fixed code, and tests that trigger the security defect. If a patch is generated that passes the test cases, it will be need to be assessed to ensure it is semantically identical to a human written patch.

There are a few papers which, albeit in much more depth, consider solely the analysis of defects with existing tools.[11] However, none so far target specifically security defects.

### 3.1.1 Tool Selection

To begin evaluation and selection, we need to find runnable, open-source tools which can take in our own programs and test suites, attempting to produce a patch which fixes the defect. Ideally, we want to find a range of tools which cover a range of different APR techniques.

#### Challenges

There were a number of tools that did not come with any functionality to input our own programs and tests [34] [35] [36], rather they exist online only to allow replication of results from an already specified set of buggy programs. Whilst there are a great many tools out there each with innovative methods and many with promising results, not being able to use them with custom code is indicative of the relative novelty of the APR field.

Additionally, a great deal of time was spent attempting to understand how these tools worked. This was in part due to my inexperience in this area, but also due to the lack of any real documentation for a large number of tools or exceptionally poor documentation for a few. Poor documentation is already an issue with open-source software, with a GitHub survey indicating 93% of developers encountered issues with incomplete of confusing documentation. [37]

#### Selection

ASTOR [17] is an APR framework which comes with 5 different types of tool (JKali[12], JMutRepair[38], JGenProg[16], DeepRepair [39], Cardumen[20]). It has functionality to take in our own programs, tests, and the selected tool to generate patches. To input programs into ASTOR, they must be packaged and built with some management tool, such as Gradle or Maven.

This was chosen to be the evaluation tool because it provides a framework to develop patches with a range of tools covering many techniques which are detailed further in Table 1. Additionally there was *some* documentation for this project, which made understanding how to use it easier.

| Strategy | Change Operator | Tool |
|---|---|---|
| General Search Based | AST reuse/insertion/deletion | jGenProg |
| General Brute Force | Functionality deletion | jKali |
| General Brute Force | Logical/Relational predicates | jMutRepair |
| General Search Based | Mined expressions | Cardumen |

Table 1: Selection of chosen tools in ASTOR

### 3.1.2 Bug Set Generation

For the chosen repair framework, the input requires a built project with all its dependencies along with the source code and test suites, where the tests are the primary method to ensure correctness of patches. Ideally, we want to work with real world defects with many lines of code as this provides genuine value in the field. Most importantly, we need to have a range of different security defects to attempt to fix for the sake of generalisability to other security defects. The sections below describes the process and findings when attempting to acquire an appropriate bug set.

#### CVEs

Vulnerabilities from the Common Vulnerability and Exposures database (CVE) were gathered, with effort made to find projects with buggy and fixed code. Unfortunately, many of these projects, as well as their third-party dependencies were deprecated. This made building them for use in our selected framework impossible, which needed functioning programs not just source code. Afterwards, we attempted to find a maintained bug set and explore this for security defects.

#### Defects4J

Defects4J[40] is a maintained database of existing faults which come from real-world projects. A bash script was written to examine information for each defect, however, no security defects were found in any of the hundreds of projects. Furthermore, the framework at the time of testing required an older version of Java which was incompatible with ASTOR. Using real-world bugs would be optimal for providing value however a different approach was needed.

#### Software Assurance Reference Dataset

The Juliet Test Suite[41] is a bug-set of thousands of automatically generated small programs in Java and C/C++ covering over 100 classes of security defects. Programs are split by Common Weakness Enumeration (CWE) and contain base cases and control variants. This suite is designed to test static analysis tools and so the programs do not actually function, however, demonstrate a good and relatively underused resource.

#### Manual Bug-set Generation

According to a recent literature survey,[2] whilst it is of great importance to utilize benchmarks, effort should be made to create benchmarks as well. From our search, we see no benchmarks for security defects exist. In order to improve the field of APR.

There are some drawbacks to using 'academic' or 'synthetic' defects such as not being representative of production code. However, these defects allows us to understand the nature of the APR tools patches more readily, and developing a bug-set will provide a framework for future research to be conducted on.

## 3.2 Automatic Test Suite Generation

Like most code generation solutions which operate on toy programs, our code generation solution will function over our bug-set of defective programs. Now that we have found a suitable bug-set of defects we can begin to examine how one might go about developing a system to automatically generate test suites for them.

### 3.2.1 Parsing ASTs from Defective Programs

ASTs are tree representations of source code, where each node represents a statement or code construct. They are abstract in the sense that they omit information that can be derived from the code, as well as comment and white space tokens, unlike a parse tree. For the purposes of analyzing files to see what information to gather, an AST will be more than sufficient.

By parsing and analyzing the AST of a defective program we can discern the information needed to write a test file. For this we will need a java parser, namely JavaParser. ANTLR was also considered, which includes grammars for an exceptional number of languages, supports parsing expression grammars, and provides facility to generate code. However, Javaparser provides methods to analyse, transform, and generate java source code and can be included as a maven dependency. It also includes a symbol solver to resolve references and find relations between nodes. An example of the AST representation the JavaParser uses is illustrated in Figure 4 and has been adapted from Tomasetti's excellent book on the JavaParser[42].

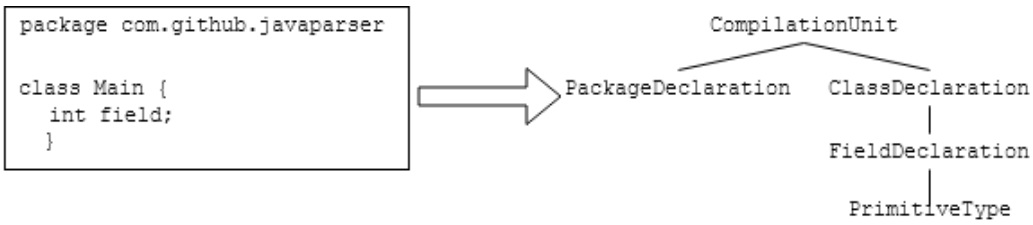


Figure 4: Sample of JavaParser AST adapted from [42]

### 3.2.2 Generating Tests

There are many approaches to generating code, each with specific requirements. A template engine for example, will take input using special notation which is then replaced with proper data at runtime. Model driven design can be used to describe the model using domain specific languages or UML diagrams. Ad hoc design encapsulates many other forms of code generation technique and relies on the idea that if you can create a description of the problem, you can solve it with code generation. For our purposes, an ad hoc approach is most suitable as this can be written solely in Java, generating a model of the problem using information parsed from the AST of the defective program. It should be noted that these tests are only designed to be exploitative, rather than testing the functionality of the program itself.

JavaPoet is an API written in java for generating .java source code files where rather than simply streaming a file out, it builds up programs as a series of declarations using a builder pattern. This is most suitable for our ad hoc approach.

### 3.2.3 Generating Test Data

Fuzzing and Taint Analysis are very popular choices for vulnerability discovery. Taint analysis tracks the flow of 'tainted' input during the execution of a program, while fuzzing involves sending data designed to cause an exploit or a crash to be triggered. Both require examination of the output. Fuzzing is a very popular choice for vulnerability discovery and is heavily used by threat actors to find exploits in programs.

We have chosen to implement a white box fuzzing approach by generating test cases that should pass for a suitably secure functions and not pass for functions containing these security defects. From here we can compare the generated tests to the manual ones, as well as the output from APR tools with generated and manual tests.

## 3.3 Evaluation

As seen from the literature, traditional test suite metrics proposed for software testing e.g. branch coverage, code coverage, have been shown to be useful when assessing how well a test suite may perform for APR.[32] For evaluating the effectiveness of our generated tests against our manually written tests we will examine:

- Method Coverage

- Line Coverage
- Execution Time

Additionally we can use the generated tests as input with SECdefects to investigate if automatically generating test suites are nearly as good as our manual test cases, and if can improve software repair by examining the output from our repair tool.

The process for evaluation of how well APR tools are guided is included below:

1. Write a defective program according to CWE with passing and failing test cases.
2. Automatically generate test cases for this program
3. Generate coverage reports and package project
4. Pass project into ASTOR using each different tool
5. Examine and document output
6. Repeat steps 1-5 for each defective program
7. Evaluate based on number and quality of patches generated.

## 3.4 Summary

In this section we have analyzed the literature, assessed the potential effectiveness of APR on security defects, and developed a testing methodology for assessing automatically generated tests against manual tests.

# 4 Design

## 4.1 Academic Bugs - SECdefects

### 4.1.1 Bugs

SECdefects is a suite of java files with example defect programs, split by CWE. It provides a useful resource by acting as a library for the analysis of APR tools.

The bugs used for the program have been selected from the CWE Top 25 most dangerous software errors.[**cwe˙team˙cwe˙nodate**]

As many defects have been included as possible but there are some limitations. Firstly, Java does not interact directly with memory some errors are impossible to replicate. Additionally, the Java Virtual Machine performs garbage collection so other errors don't need including. Finally, the use of .WAR files is incompatible with the APR tools of choice, so errors requiring JavaServer Pages are not included.

- CWE-20: Improper Input Validation
- CWE-22: Path Traversal
- CWE-89: SQL Injection
- CWE-125: Out-of-bounds Read
- CWE-200: Information Exposure

### 4.1.2 Requirements

A list of software requirements based on MoSCoW priority have been included below. As this program is primarily for use in research certain non-functional requirements have been omitted. According to a research project by Paul E. Black,[41] the ultimate test suite has three aspects: represents production software, we know where all the bugs are, and it has lots of different types of bugs in varied situations. These three aspects have been taken into consideration when deciding upon the following requirements in Table 2.

| MoSCoW Priority | Requirement |
|---|---|
| Must | Include decided-upon security defects |
| Must | Define security defects so we know where they are |
| Must | Include functional and exploitative test cases |
| Must | Package and pass defective program into an APR tool |
| Must | Be able to specify a certain CWE for repair |
| Could | Include alternative control flow variants for each defect |
| Could | Include alternative data flow variants for each defect |
| Won't | Automatically compile defective program, passing into APR tool |

Table 2: Requirements of suite

Our cases are far smaller and less complex than production ready code, however they aim to test a variety of different bugs, as well as including bugs in known 'bad' functions.

## 4.2 Automatic Test Generation - Security Suite Generator

### 4.2.1 Specification

Our Security Suite Generator (SSG) is designed to work over the bug-set of SECdefects by automatically generating compilable JUnit test cases for a specified defect. As such the pieces of information we need to gather from our defective program are:

- Class Declaration
- Package Declaration
- Method Declarations

From here, we can build a class report and send this to a test builder. Our test builder is designed to read a class report, generate a skeleton JUnit class, and fill this with the relevant tests before writing it out to a .java file. JavaPoet automatically handles all imports for us, provided we give the information as a type literal and provides functionality to write out our class as well.

A flowchart detailing these system processes visually is illustrated in Figure 5.

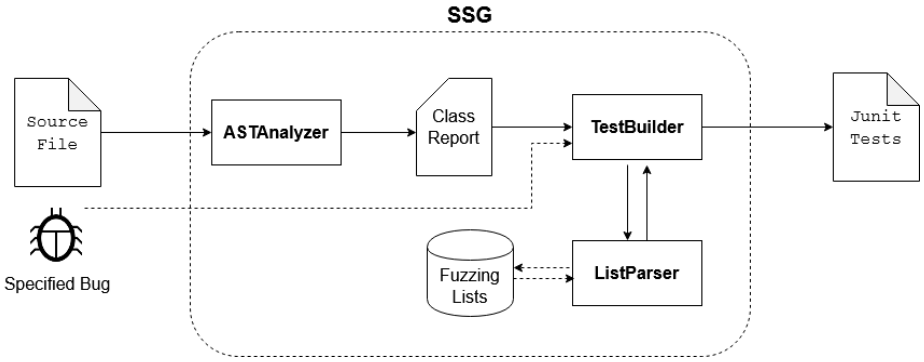


Figure 5: Proposed system flowchart for SSG

### 4.2.2 Fuzzing Lists

Open source fuzzing lists were examined and ultimately fuzzing lists 1N3@Crowdshield's Intruder Payload list, [43] was chosen as our base and is free to use and modify. Our lists were adapted by choosing the most commonly used payloads, taking out extensively malformed strings, and reducing them in number slightly as we had to keep in mind that test suites are by far the dominant factor in APR tools.[32] These were then parcelled into 3 .csv files for integer attacks, SQL attacks, and path traversal attacks.

### 4.2.3 Requirements

A requirements analysis was conducuted for SSG, included in Table 3. Again, as this is mainly designed for research purposes, certain non-functional requirements have been omitted.

| MoSCoW Priority | Requirement |
|---|---|
| Must | Parse a program to produce an AST |
| Must | Extract relevant information from a program's AST |
| Must | Parse a series of payloads from a fuzzing list |
| Must | Generate test cases from a fuzzing list for a given program |
| Must | Generate compilable JUnit tests suites for a program |
| Must | Be able to generate test cases for each defective program in SECdefects |
| Should | Allow a user to specify a program and defect |
| Could | Automatically generate all possible test suites |

Table 3: Requirements of SSG with MoSCoW prioritization

## 4.3 Summary

In this section we have defined more explicitly what our programs will do, providing a requirements table for each.

# 5 Implementation

## 5.1 SECdefects

SECdefects is our designed suite of defective code. Code is split by CWE. Each defective program contains buggy code, fixed code, and test cases. Both passing functional tests, and failing exploitative tests have been included to enable to evaluation of APR tools.

The program was written in Java using JDK8 to conform with ASTOR's use of JDK8 and tests written using JUnit 4.11. Dependencies and packaging are handled with Maven, with GitHub used for Version Control. Programs were written using a test-driven development process and have been tested on Windows 10 and Ubuntu 16.04.

### 5.1.1 Prototyping

To test the process of producing patches with ASTOR an example from `apache.commons.math` was used. A patch was successfully produced as seen in Figure 6); however, the output was printed on one line and was very difficult to read, highlighting the need to check APR output carefully.


Figure 6: Example APR tool output

Next, a sample program for CWE-20 was developed, with a minimal, exploitative test case for only the bad code. This was passed into the APR tool which successfully produced a patch for the problem.

Progress could then be made and class diagrams were drawn up for the required classes, see Fig 7.

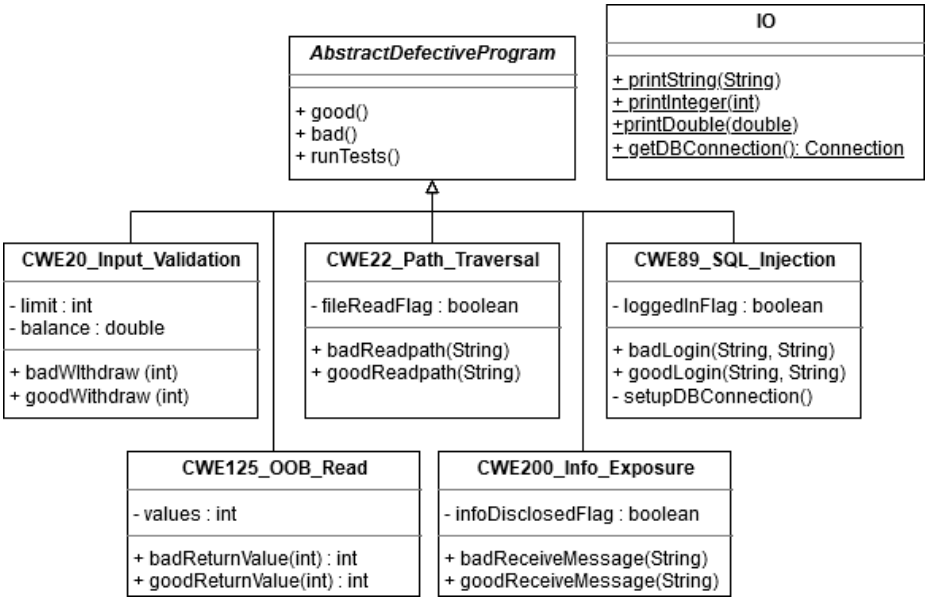


Figure 7: UML class diagrams for SECdefects

### 5.1.2 Changes from Proposed Design

Due to time constraints, it was decided not to pursue completion of CWE-200 after concluding that the 4 written defects would be sufficient. Additionally, following execution of generated tests, we uncovered some additional security issues with our initial design which required us to revise it. Changes made to SECdefects are:

- Manual tests split into Functional and Exploitative sections.
- CWE-22: Added URL decoding for path traversal after generated SSG test suites caused an unforeseen error.

### 5.1.3 Structure

The structure includes an explanation of each class and a description of the exploit. Effort has been made to make the defects as simple as possible to assess the current ability of repair tools.

Each defective program has a working, defect-free function which is referred to as *good code*. Functions or sections of code containing a security defect are referred to as *bad code*. With each program, a series of identical test cases have been included, for relevant good and bad functions only.

A Javadoc has been included and can be found under the `javadoc` directory.

#### CWE-20

Contains a very basic improper input validation defect, by simulating a bank transaction. If a withdrawal input is negative, and incorrectly validated, arbitrary positive balance can accrue.

```
/* FIX only allow positive quantities by including another validation of input*/
if (amount < 0 || amount >= LIMIT)
    return;
balance = balance - amount;
```

Figure 8: Good code for CWE-20

```
/* FLAW negative quantity of orders should not be allowed otherwise attacker
* can increase balance arbitrarily*/
if (amount >= LIMIT)
    return;
balance = balance - amount;
```

Figure 9: Bad code for CWE-20

### CWE-22

Users supply a profile.txt they wish to view, kept under the profiles directory. However there is no neutralization of special characters. This enables potential path traversal to another folder under the same resources directory, called passwords and a password.txt which can be read.

By assessing the absolute path against the canonical path (which striped paths of special characters), we can detect if any potential file traversal is taking place.

```
String relativeFilepath = "src/main/resources/profiles/" + inputFilepath;

// Perform Double URL decoding
try {
    relativeFilepath = URLDecoder.decode(
            URLDecoder.decode(relativeFilepath, StandardCharsets.UTF_8.name()), StandardCharsets.UTF_8.name());
} catch (Exception e) {
    e.printStackTrace();
}

File file = new File(relativeFilepath);

/* FIX check absolute against canonical path where canonical disregards special characters*/

String canonicalFilepath = null;
String absoluteFilepath = null;

try {
    canonicalFilepath = file.getCanonicalPath();
    absoluteFilepath = file.getAbsolutePath();
} catch (Exception e) {
    e.printStackTrace();
}

if (canonicalFilepath == null || absoluteFilepath == null)
    return;

if (!canonicalFilepath.equals(absoluteFilepath)) {
    IO.printLine("Potential Directory Traversal");
    return;
}

BufferedReader in = null;

try {
    in = new BufferedReader(new InputStreamReader(new FileInputStream(file), StandardCharsets.UTF_8));
```

Figure 10: Good code for CWE-22

```java
public void badRead(String inputFilepath) {

    String relativeFilepath = "./src/main/resources/profiles/" + inputFilepath;

    // Perform Double URL decoding
    try {
        relativeFilepath = URLDecoder.decode(
                URLDecoder.decode(relativeFilepath, StandardCharsets.UTF_8.name()), StandardCharsets.UTF_8.name());
    } catch (Exception e) {
        e.printStackTrace();
    }

    File file = new File(relativeFilepath);

    /* FLAW code doesn't limit the potential input for the file path */

    BufferedReader in = null;

    try {
        in = new BufferedReader(new InputStreamReader(new FileInputStream(file), StandardCharsets.UTF_8));
```

Figure 11: Bad code for CWE-22

### CWE-89

This Program allows the user to login with a provided username and password which is searched against table of usernames and passwords.

The database was created as an in-memory database with SQLite to reduce complexity of including a separate database program, although a .keep file was placed under the resources folder for Git to version it, which is empty until this program is executed.

```java
// FIX Code uses prepared statements for dynamic queries
String sql = "SELECT * FROM users WHERE username=? AND password=?";

PreparedStatement preparedStatement = connection.prepareStatement(sql);
preparedStatement.setQueryTimeout(10);

preparedStatement.setString( parameterIndex: 1, username);
preparedStatement.setString( parameterIndex: 2, password);

ResultSet rs = preparedStatement.executeQuery();
```

Figure 12: Good code for CWE-89

```java
/* FLAW Code uses dynamically created statements from user inputs */
String sql = "SELECT * FROM users WHERE username = '" + username + "'AND password = '" + password + "'";

Statement statement = connection.createStatement();
statement.setQueryTimeout(10);

ResultSet rs = statement.executeQuery(sql);
```

Figure 13: Bad code for CWE-89

### CWE-125

A simple example of an out-of-bounds read on an array of a fixed size given certain input. Unlike the other defects, this one does cause interruption of program flow by throwing an `ArrayIndexOutOfBoundsException`.

```java
int value;

/* FIX Include a check to ensure array index is within boundaries*/
if (index >= 0 && index < aValues.length) {
    value = aValues[index];
} else {
    System.out.println("Error bad value");
    value = -1;
}

return value;
```

Figure 14: Good code for CWE-125

```java
int value;

/* FLAW This will allow a negative value to be accepted as the input array index*/
if (index < aValues.length) {
    value = aValues[index];
} else {
    System.out.println("Error bad value");
    value = -1;
}

return value;
```

Figure 15: Bad code for CWE-125

## 5.2 SSG

### 5.2.1 Structure

An explanation of the implemented classes is included below. A JavaDoc has been included with the design archive, along with a README file.

- **Main:** The main class allows user interaction with command line, allowing a user to specify a source code file as well as a defect type. Main also controls program flow to read reports, parse the appropriate lists and finally build the desired test suite.
- **ClassAnalyzer:** Using the JavaParser API we can analyze the AST of a program by defining a Visitor classes which step through our AST and collect necessary information from the specification declarations. From that, we generate a Class Report
- **ClassReport:** An abstraction capturing the relevant information from our source code file in order to build our JUnit test suite. This includes, package name, class name and all method names.
- **DefectType enum:** Indicates the different defects that Test Suites can be generated for. Information on which parameter is included at the start of the name of each DefectType to indicate which parameter the associated good/bad methods should take. So far, these include: Integer Attacks, Validation Attacks, SQL Injection Attacks, and Path Traersal Attacks
- **ListParser:** Class to Parse payload lists from under the src/main/resources/lists which provides the data for generating JUnit test cases
- **TestBuilder:** Generates JUnit Test Suites based on report read and defect type chosen. Each test case covers a different input

### 5.2.2 Specification of Inputs

**Input File** The generator is built for use with SECdefects, however, as we are parsing the AST from a program it does not matter what the specific contents are. As long as any input program follows the specification laid out below, we can generate test suites.

- File is compilable
- File includes package declaration
- File includes class declaration
- File includes methods prefixed with 'good' OR 'bad' with appropriate parameters
- Optional: method prefixed with 'is' to be used as oracle function is included

Two example files have been included in the design archive `ExampleInteger.java` and `ExampleString.java` meet the minimum design specification for their appropriate methods. These have been included below in Figure 16 and Figure 17.

```java
package com.example;

public class ExampleInteger {

    public void badExample(int input) {}

    public void goodExample(int input) {}

}
```

Figure 16: Example minimal specification for integer test generation

```java
package com.example;

public class ExampleString {

    private boolean oracleFlag = false;

    public void badFunction(String input) { }

    public boolean isOracleLogin() {
        return oracleFlag;
    }
}
```

Figure 17: Example minimal specification for string generation

**Defect Type** The program will assess vulnerabilities in five different situations. Each type corresponds to which kind of fuzzing lists to use, and tells our test builder what sort of method parameters we should be expecting:

- INTEGER_ATTACK: Takes in 1 Integer parameter e.g. `badCheckIndex(int index)` passing in common integer attack payloads.
- INTEGER_VALIDATION: Takes in 1 Integer parameter specifically for CWE-20 of SECdefects i.e. `badWithdraw(int amount)` passing in common integer attack payloads.

- STRING_PATH_TRAVERSAL: Take in 1 String parameter ideally for CWE-22 of SECdefects i.e. `badReadFile(String path)` passing in common path traversal attack payloads.
- STRING_SQL_INJECTION: Takes in 1 String parameter e.g. `badQuery(String query)` passing in common SQL Injection payloads.
- STRING_SQL_INJECTIONS: Takes in 2 String parameters e.g. `badLogin(String username, String password)` passing in common SQL Injection payloads.

### 5.2.3 Specification of Output Program

The program will generate JUnit test suites for each defect. Each test case will fuzz a different input. It is important to note that the tests are almost compilable - a small issue with JavaPoet in which initialization specifications are surrounded with {}, which must be removed prior to our test suite being compiled.

Whilst we will use the output of programs from SECdefects for our evaluation and testing, if we pass in our example programs we get the following output in Figure 18 and Figure 19. These screenshots do not included the whole programs, which are 500+ and 700+ lines long respectively.

```
package com.example;

import ...

/**
 * AUTOMATICALLY GENERATED TEST SUITE
 * NOTE: TO COMPILE CORRECTLY PLEASE REMOVE { } AROUND INITIALIZATION FIELD AT TOP
 * @author cc14g17
 */
public class ExampleIntegerAUTOGEN_EXPLOIT_Test {
    {
        private ExampleInteger exampleinteger;
    }

    @Before
    public void setUp() { exampleinteger = new ExampleInteger(); }

    @Test
    public void badExample1() { exampleinteger.badExample(0); }

    @Test
    public void badExample2() { exampleinteger.badExample(1); }

    @Test
    public void badExample3() { exampleinteger.badExample(2); }
```

Figure 18: Test generation for example integer file

```java
package com.example;

import ...

/**
 * AUTOMATICALLY GENERATED TEST SUITE
 * NOTE: TO COMPILE CORRECTLY PLEASE REMOVE { } AROUND INITIALIZATION FIELD AT TOP
 * @author cc14g17
 */
public class ExampleStringAUTOGEN_EXPLOIT_Test {
    {
        private ExampleString examplestring;
    }

    @Before
    public void setUp() { examplestring = new ExampleString(); }

    @Test
    public void badFunction1() {
        examplestring.badFunction("OR 1=1");
        Assert.assertFalse(examplestring.isOracleLogin());
    }

    @Test
    public void badFunction2() {
```

Figure 19: Test generation for example string file

### 5.2.4 Changes from Proposed Design

**DefectType enum:** During the course of development, it was apparent that using an enumerated type to choose the defect type would be more appropriate than anything else. Additionally, it contributes towards maintainability if additional defects are added for generation.

**Generating Assertions:** When the respective security defect for our programs are triggered, only one of our programs will throw an error, namely CWE-125 which throws an `ArrayIndexOutOfBoundsException`. For our other programs, our test cases need some way of asserting whether the intended vulnerability has been triggered, namely a testing oracle, which ensures a program behaves to some specification. For each program our oracle is different, but the inclusion of an `oracle function` was used i.e. a get method for our flag field in SECdefects programs. More specifically, assertions were generated by:

- CWE-20 Asserts what the expected value should be by knowing intended functionality of program. Does not require specific oracle function.
- CWE-22 Requires an oracle stating whether or not a file has been read. Requires minimal context of program's file structure by append the path traversal payloads with files we know exist, that are read if the defect is exploited.
- CWE-89 Requires an oracle to tell us if a login has been successful. This will allow us to assess whether a payload allows a user to log in without proper credentials. No knowledge of the program is required.
- CWE-125 No oracle required as payloads are designed to interrupt program flow and cause an exception.

Essentially, test suite generation for CWE-89 and CWE-125 can be readily applied to other programs which fit the specification, whereas CWE-20 and CWE-22 require some knowledge about the program itself.

## 5.3 Summary

In this section we have covered implementation details for our suite, SECdefects, and our test suite generator, SSG.

# 6 Testing

## 6.1 SECDefects

### 6.1.1 Testing against requirements

| Priority | Requirement | Met? |
|---|---|---|
| Must | Include decided-upon security defects | Partial |
| Must | Define security defects so we know where they are | Yes |
| Must | Include functional and exploitative test cases | Yes |
| Must | Package and pass defective program into an APR tool | Yes |
| Must | Be able to specify a certain CWE for repair | Yes* |
| Could | Include alternative control flow variants for each defect | No |
| Could | Automatically compile defective program, passing into APR tool | No |

Table 4: Requirements testing of SECdefects

Requirement 1 is partially met due to only 4 out of the desired 5 defective programs being developed due to time constraints. Additionally, for requirement 5, ASTOR allows one to specify a failing test case. As long the other tests are not included, as ASTOR uses all tests for regression purposes, ASTOR will function correctly.

### 6.1.2 Manual Test Suites

To assess that our tests function correctly i.e. validates the security defects that exist in the program, we should expect our only our exploitative tests for our bad methods to fail, which were received and are shown in Figure 20 after running `mvn clean package`


Figure 20: Test cases fail successfully

## 6.2 SSG

### 6.2.1 Testing Against Requirements

| Priority | Requirement | Met? |
|---|---|---|
| Must | Parse a program to produce an AST | Yes |
| Must | Extract relevant information from a program's AST | Yes |
| Must | Parse a series of payloads from a fuzzing list | Yes |
| Must | Generate test cases from a fuzzing list for a given program | Yes |
| Must | Generate compilable JUnit tests suites for a program | Partial |
| Must | Be able to generate test cases for each defective program in SECdefects | Yes |
| Should | Allow a user to specify a program and defect | Yes |
| Could | Automatically generate all possible test suites | No |

Table 5: Requirements testing of Security Suite Generator

JavaPoet includes braces around initialization, hence the partial requirement. Taking out these braces gives a fully compilable program. Generated test classes include an annotation explaining this problem and fix at the top.

### 6.2.2 Test Suite Generation

After running the program on SECdefects, we successfully generated hundreds of test cases for each defective class. All good methods successfully passed all test cases, whilst a number of bad methods failed some tests and can be seen in Figure 21.

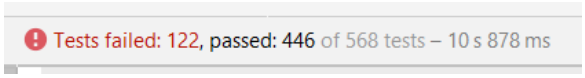


Figure 21: Results from automatically generated test suites

From this, we can see can see that automatic test suite generation using fuzzing to generate test data is an effective approach to finding defects that exist within programs. For all our defective programs, additional exploit strings were found. Additionally, we had to redo our CWE-22 program as URL encoded strings caused unintended. Truncated output figures of our generated tests have been included in Appendix B.

## 6.3 Test Suites

Below are the results from the code coverage metrics of manually written test cases, generated test cases, and generated test cases with functional manual tests. It should be noted that 100% coverage should not be expected, the tests are only written to exploit our relevant good and bad methods.

| CWE ID | Method (%) | Line (%) |
|---|---|---|
| 20 | 57 | 68 |
| 22 | 71 | 66 |
| 89 | 62 | 70 |
| 125 | 60 | 68 |

Table 6: Code coverage for manual tests

| CWE ID | Method (%) | Line (%) |
|---|---|---|
| 20 | 57 | 68 |
| 22 | 57 | 51 |
| 89 | 62 | 68 |
| 125 | 60 | 78 |

Table 7: Code coverage for generated tests

| CWE ID | Method (%) | Line (%) |
|---|---|---|
| 20 | 57 | 68 |
| 22 | 71 | 69 |
| 89 | 62 | 72 |
| 125 | 60 | 78 |

Table 8: Code coverage for generated & functional manual tests

Higher coverage is also not naturally indicative of a better test, but we can see that automatically generated test suites for the most part perform nearly as well as our manual tests. Additionally, from Figure 21 we see that our generated tests discover hundreds of inputs that trigger our defective code. Although some of these may not be entirely relevant, it still demonstrates a useful methods for detection of security defects.

Our generated tests seem to perform best where when a great deal of context about the original program is not required i.e. CWE-125. For CWE-20, we would have needed to know the files to search for in the file directory to trigger the oracle and so performs less well, although a more precise oracle method may provide better results.

A combination of automatically generated and manual functional tests however shows the best coverage. Bearing in mind that test suite metrics can function as guidelines for assessing how well a solution can be found, we can hypothesise that these tests may guide APR tools better than just manual tests, which we will examine next.

## 6.4 Summary

In this section we tested our programs against their specified requirements to ensure they function correctly, and provided a comparison on different testing approaches examining their traditional test suite metrics.

# 7 Evaluation

## 7.1 Findings

First we will examine the difference our testing suites have made, and then we will examine the patches produced by our APR tools.

### 7.1.1 Patch Generation

After passing in our defective programs from SECdefects and test suites into ASTOR with the selected tools, we receive the following outputs in Tables 9, 10, and 11. Rows have been omitted if all programs were unsuccessful in finding a solution. A dash '-' indicates no solution was found

Information from each output was noted, specifically: number of semantically-correct solutions found, number of semantically-incorrect found, and time taken to complete in seconds

| | jGenprog | jKali | jMutRepair | Cardumen |
|---|---|---|---|---|
| CWE-20 | 1, 2, 35.3 | - | - | 1, 0, 9.10 |
| CWE-125 | - | - | - | 0, 1, 17.9 |

Table 9: Results from APR tools using manual tests (functional & exploitative)

Only solutions for CWE-20 were discovered with manual tests, whilst a non-semantically correct patch was produced for CWE-125. Our tests concluded in a short amount of time for both programs.

| | jGenprog | jKali | jMutRepair | Cardumen |
|---|---|---|---|---|
| CWE-20 | 1, 1, 143 | - | - | 1, 0, 21.2 |
| CWE-125 | 0, 1, 47.7 | - | - | 0, 5, 23.0 |

Table 10: Results from APR tools using generated tests

We can confirm very quickly that test suites are the most influential factor for run time in APR tools. Many more semantically incorrect solutions were found for CWE-125 whilst still finding solutions for CWE-20. Additionally, a semantically incorrect solution was produced for CWE-125 with jGenProg.

| | jGenprog | jKali | jMutRepair | Cardumen |
|---|---|---|---|---|
| CWE-20 | 1, 1, 61.7 | - | - | 1, 0, 14.5 |
| CWE-125 | - | - | - | 0, 1, 15.1 |

Table 11: Results from APR tools using generated & functional manual tests

We do not find any new patches, nor does our performance in finding more semantically correct solutions improve. However, including functional tests significantly reduces the time taken for our APR tools to finish, even with hundreds more test cases. This is because functional tests provide a tighter specification of the problem to solve for, so incorrect solutions are discarded more quickly.

#### Examination of Tools

No solution to any problem was solved by jMutRepair or jKali. Both of these program's change operators are far too simple to solve any security defect. No solutions were found for CWE-22 nor CWE-89, evidently the search space required to solve these problems is too great. jGenProg and Cardumen perform best at finding solutions, with Cardumen finding more solutions more quickly which can be attributed to having the largest set and most relevant set of change operators.

### Examination of Results

More semantically incorrect solutions were produced using solely our generated test cases. The fitness function for each candidate solution is calculated from the number of passing and failing test cases, and so having a looser specification means more potential candidates are explored.

We can see that when it comes to developing semantically correct patches for a program, generated tests perform just as well as manual tests. By automatically generating test suites, we include a good number of relevant, failing test cases which has shown improvement to APR in previous research. No new patches were found, however, this may be due to the limitations of the tools themselves, rather than the tests.

### 7.1.2 Patch Examination

**CWE-20** The same program successful was produced for CWE-20 by both jGenProg and Cardumen and is semantically correct i.e. the same as our 'good' code (see Figure 22).

```
--- src/main/java/ccl4gl7/CWE20_Improper_Input_Validation.java
+++ src/main/java/ccl4gl7/CWE20_Improper_Input_Validation.java
@@ -21,9 +21,9 @@
    }

    public void badWithdraw(int amount) {
-       if (amount >= LIMIT)
+       if ((amount < 0) || (amount >= LIMIT)) {
            return;
-
+       }
        balance = balance - amount;
    }
```

Figure 22: Successful patch for CWE-20

**CWE-22** No correct solutions were produced for this program, which is to be expected as the patch for this program is quite complex.

```
cal@wang2200: ~/ip/tools/astor/astor
File  Edit  View  Search  Terminal  Help
[INFO ] fr.inria.astor.core.setup.ConfigurationProperties.print(ConfigurationProperties.java:71) - ---------------------------
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:59) - mod_point MP=cc14g17.CWE22_
Path_Traversal line: 67, pointed element: CtIfImpl
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:60) - -->op: OP_INSTANCE:
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) `in != null ` -topatch--> `in == null` (spoon.support.reflect.code.CtBinary
OperatorImpl)
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:59) - mod_point MP=cc14g17.CWE22_
Path_Traversal line: 67, pointed element: CtIfImpl
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:60) - -->op: OP_INSTANCE:
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) `in != null ` -topatch--> `in >= null` (spoon.support.reflect.code.CtBinary
OperatorImpl)
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:59) - mod_point MP=cc14g17.CWE22_
Path_Traversal line: 67, pointed element: CtIfImpl
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:60) - -->op: OP_INSTANCE:
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) `in != null ` -topatch--> `in > null` (spoon.support.reflect.code.CtBinaryO
peratorImpl)
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:59) - mod_point MP=cc14g17.CWE22_
Path_Traversal line: 67, pointed element: CtIfImpl
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:60) - -->op: OP_INSTANCE:
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) `in != null ` -topatch--> `in <= null` (spoon.support.reflect.code.CtBinary
OperatorImpl)
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:59) - mod_point MP=cc14g17.CWE22_
Path_Traversal line: 67, pointed element: CtIfImpl
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:60) - -->op: OP_INSTANCE:
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) `in != null ` -topatch--> `in < null` (spoon.support.reflect.code.CtBinaryO
peratorImpl)
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:59) - mod_point MP=cc14g17.CWE22_
Path_Traversal line: 67, pointed element: CtIfImpl
[INFO ] fr.inria.astor.core.solutionsearch.ExhaustiveSearchEngine.startEvolution(ExhaustiveSearchEngine.java:60) - -->op: OP_INSTANCE:
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) `in != null ` -topatch--> `!
                (in != null)` (spoon.support.reflect.code.CtUnaryOperatorImpl)
[INFO ] fr.inria.astor.core.solutionsearch.AstorCoreEngine.atEnd(AstorCoreEngine.java:171) - Time Repair Loop (s): 1.239
[INFO ] fr.inria.astor.core.solutionsearch.AstorCoreEngine.atEnd(AstorCoreEngine.java:173) - generationsexecuted: 1
[INFO ] fr.inria.astor.core.solutionsearch.AstorCoreEngine.printFinalStatus(AstorCoreEngine.java:248) -
---SUMMARY_EXECUTION---
[INFO ] fr.inria.astor.core.solutionsearch.AstorCoreEngine.printFinalStatus(AstorCoreEngine.java:258) - End Repair Search: NOT Found solution
[INFO ] fr.inria.main.evolution.AstorMain.run(AstorMain.java:190) - Time Total(s): 6.247
cal@wang2200:~/ip/tools/astor/astor$
```

Figure 23: No solution output for CWE-22

**CWE-89** No correct solutions were produced, however we can see from Cardumen's `ingredients.json` (mined expressions) that `PreparedStatements` are collected, see Figure 24. This suggests that the fix for the program does exist in the large search space. With optimizations a solution may be found but currently a fix is intractable.

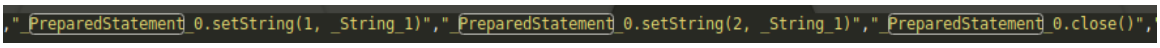


Figure 24: Examination of mined statements in CWE-89

**CWE-125** No semantically correct solutions were produced. Many solutions were found that simply deleted whole sections of code altogether are is demonstrative of some of the limitations of APR tools.

```
--- src/main/java/cc14g17/CWE125_Out_of_Bounds_Read.java
+++ src/main/java/cc14g17/CWE125_Out_of_Bounds_Read.java
@@ -23,7 +23,7 @@
    public int badGetValue(int index) throws java.lang.ArrayIndexOutOfBoundsException {
        int value;
        if (index < aValues.length) {
-           value = aValues[index];
+           value = index++;
        } else {
            java.lang.System.out.println("Error bad value");
            value = -1;
```

Figure 25: Semantically-incorrect patch for CWE-125

## 7.2 Static Analysis

Static analysis using findbugs only find 1 out of 4 of the security defects, as can be seen in 26 with the full output in Appendix C. Whilst static analysis is an excellent tool, it is not able to identity more subtle security defects, whereas our generated tests are able to identify numerous failing cases see Figure 21.

**Warnings**

Click on each warning link to see a full description of the issue, and details of how to resolve it.

**Security Warnings**

| Warning | Priority | Details |
|---|---|---|
| Nonconstant string passed to execute or addBatch method on an SQL statement | High | cc14g17.CWE89_SQL_Injection.badLogin(String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement<br><br>In file CWE89_SQL_Injection.java, line 51<br>In class cc14g17.CWE89_SQL_Injection<br>In method cc14g17.CWE89_SQL_Injection.badLogin(String, String)<br>At CWE89_SQL_Injection.java:[line 51]<br>At CWE89_SQL_Injection.java:[line 51] |

Figure 26: Results of findbugs static analysis

## 7.3 Reflection on Project Goals

We will now analyze the goals of the project by reflecting on our original research questions:

- **RQ1:** How effective are current APR techniques regarding security defects?

  For this, we performed an extensive investigation into the current literature to understand the different techniques APR tools use. Upon analysis of the literature, we realized that no suites currently exist and so designed and implemented our own. After passing these into APR tools, we can conclude that a few APR techniques have the potential to fix a few simple, general security defects. However, most remain far too complex.

- **RQ2:** How feasible is Automatic Test Suite Generation for improving program repair?

  We examined different ways to automatically generate tests and test data, settling on an ad hoc fuzzing approach. We proposed a system to generate tests for our own bug-set as a proof-of-concept work. From the coverage metrics and APR tool output, although largely empirical, demonstrate generated tests to be as effective as manually written tests.

- **RQ3:** What are the contributions we can make towards improving program repair for security defects?

  We have learnt and shown value in producing a suite of defects to test APR tools, as well as provide evidence on the usefulness, and potential uses, of automatically generated test suites. Both programs have been designed with extensibility in mind for potential future research.

## 7.4 Summary

In this section, we passed our defective programs into APR tools with manual and generated test suites and dissected the output. This revealed it is unlikely APR tools are able to solve security defects.

# 8 Project Management

## 8.1 Process

An agile approach used for development, with test-driven development being utilized where possible. A Kanban-style approach to keeping track of tasks was used by splitting tasks up into to-do, doing, done, and blocked sections. For this purpose Trello used initially; however, a manual approach of using post-its on a wall was adopted after listening to a podcast by The Changelog, [44] which suggested that using agile management tools without an experienced understanding of the agile process limits their effectiveness. Using post-its proved to be a more successful way to keep track of the various stages of the project. Tasks were colour coded by size and then broken down into various sub-tasks when required. The task board wall is shown in Figure 27.

Overleaf was used for report writing, with Zotero as the reference management. Both these proved very useful and will be utilized in the future. Sections of the report were drafted as the project went along, which made writing the final report less challenging.


Figure 27: Kanban-style task board

## 8.2 Progress

The GANTT chart proposed in the progress report has been included, see Figure 28.

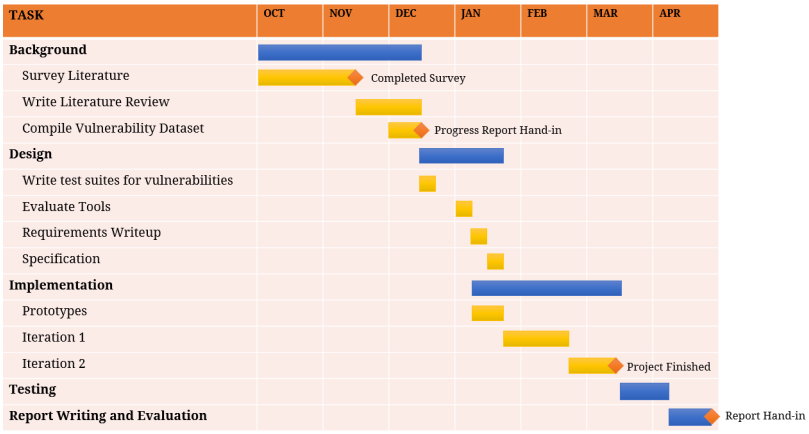


Figure 28: GANTT chart from progress report

A GANTT chart of the project's actual schedule is shown in Figure 29.

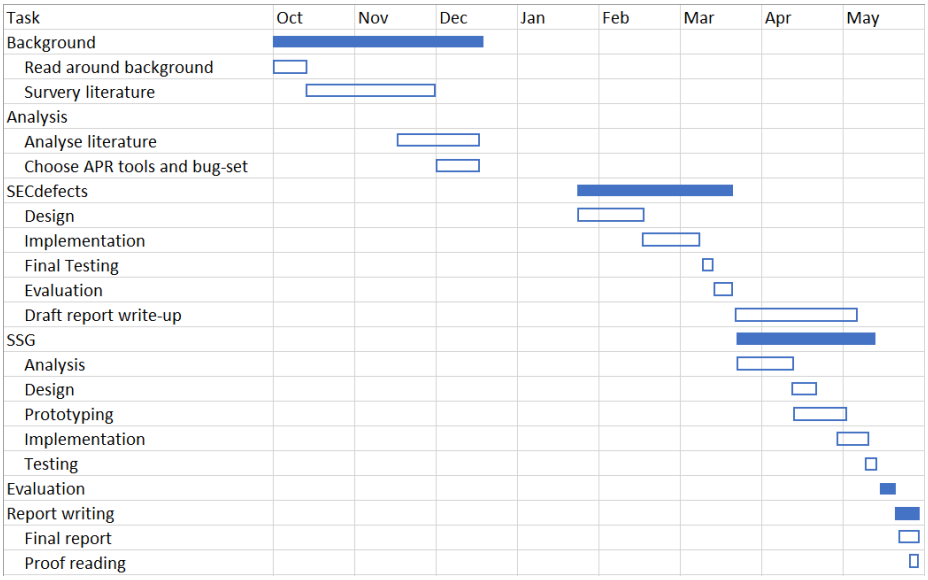


Figure 29: GANTT chart of realised progress

As we can see, the project largely followed the process outlined in the original GANTT chart. A great deal of time was spent reviewing the literature, which is required for a project in this area. However, too much time was spent in this area and going forward, it will be kept in mind to move to design and implementation as soon as possible. Development of SSG was shorter than SECdefects as a solid development workflow had been found. Keeping documentation throughout aided in writing the draft report, and final report.

Our contingency plan from the progress report has been included in Appendix A. A copy of the original project proposal, has been included in Appendix D. We have largely met the goal laid out in the project brief, with the exception of utilizing production defects. However, we have extended our goal by including a suite of security defects. Upon reflection, the goal aimed at in the project brief was too broad in scope. Delimiting the scope earlier on, perhaps to focus on a particular defect, would have been helpful. This is something that will be kept in mind for future projects.

## 8.3 Challenges

As this this is a very new area of research there were several challenges that came with it. Trying to work with poorly documented tools, deprecated projects, or closed-source, non-publicly available tools was challenging and in parts demotivating. Additionally, due to COVID-19 there were additional stresses, as well as a lack of University services which were heavily utilized previously. This lengthened the project somewhat. An extension of 2 weeks was requested and granted which helped aid the situation.

# 9 Conclusion

## 9.1 Effectiveness of APR Tools for Security Defects

After covering a range of security defects with some of the most common general APR tools, we can see that only a fraction of rather simple security defects are successfully fixed by the tools. Even for simple programs such as CWE-125, APR tools will not necessarily provide a semantically correct fix. At this stage, any output will still require a manual analysis.

For some tools, their atomic change operators are too simple to ever provide a fix - the solution to the problem does not exist in their search space. For programs such as Cardumen which use example-based techniques, the solutions to security defects do exist in their search space. Unfortunately, this search space is extremely large and because of this finding a fix for complex security defects is currently an intractable problem. Template-based methods, or defect-specific example-based methods, would be most effective.

Ideally, we would have liked to assess how effective APR tools are at repairing memory related issues, such as buffer overflows, in languages such as C. This would also let us assess the effectiveness of angelic fixes. However, these already have specific repair solutions and C is not a proficiency of mine.

As these results come from deliberately simple representations of defects, production code which contains these same defects will naturally be more complex and complicated to produce a patch for. Therefore, we can conclude that the majority of security defects are too complex in nature for general APR techniques to solve.

## 9.2 Test Suite Generation

Code generation is an effective technique for solving many problems and keeps a single source of truth. In SSG, we can generate hundreds of test cases quickly, readily expanding the fuzzing lists as required, and cover a range of defects. The generated test cases are very useful for detecting defects and, when compared to manually written tests show promising metrics. Regarding guiding program repair, our results are rather inconclusive. They perform as well as manually written tests but no not find new patches; however, this is more than likely a result of the limitations of the APR tools as opposed to our tests. A combination of functional and generated tests provides a tighter specification on the program to be generated, and the best metrics.

We have learnt a number of things during the development of our test suite generator. Test case generation using security specific inputs can be a useful technique, finding defects in programs that were not originally uncovered by manual inspection nor static analysis. Programs that do not require the use of a way to evaluate the correct behaviour of a program with a given input, known as a test oracle.

A test oracle allows us to make assertions about the program under examination and generate whole test suites. It has been stated in some papers,[45] that requirement of a testing oracle for detecting security defects is not necessary;however, we have found this not to necessarily be true. Security defects are often exploited while the program is executing normally whilst not violating any of it's functional specifications whatsoever. Indeed, only one only 1 of our 4 programs will throw an error when the defect is triggered. Going forward, we can conclude that test suite generation for security defects will require the use of an oracle, and the effectiveness of that oracle is important.

## 9.3 Further Work

An extension of the analysis tool which provides further evidence on the effectiveness of APR for security defects would be beneficial. There are a number of ways in which one can improve upon our analysis tool: adding additional tests to create more suites for the most critical defects, testing the suite on an increased number of tools, extending the testing suites to examine the impact weak test suites have on APR.

The most substantial improvement that can be made for SSG would be to extend it to production code. This would require two things. Firstly, in production code we do not know where the defects

lie. Using the JavaSymbolSolver we can resolve references in the code and assess where inputs are taken in to a program. Secondly, we do not know what the defective program takes in as inputs. Further automation of the generator to analyse function parameter information and deduce the most appropriate tests to use would be beneficial.

Furthermore, using taint analysis for test data generation would allow us more precise tracking of erroneous inputs. This would reduce the need for hundreds of fuzzing test cases, substantially reducing time overheads.

Although the oracle problem is difficult and deriving the intended functionality from any given program is problematic; however, solving it would make extending test generation to any number of problems much more feasible. Being able to derive an oracle function from a given program would allow our test cases to generate assertions, which would be incredibly useful for solving security defects, and indeed many other defects.

A very interesting idea (credit to Dr. Rathke) considers the use of security policies in place of a testing oracle for generation of testing suites. Security policies are much more coarsely grained than functional test oracles and are much more relevant to the problem at hand. From our findings, a policy-based approach to test suite generation would be exceptionally useful.

Finally, semantics-based repair seems to be the most promising avenue for repairing security defects. This techniques does not rely on other APR tools but requires specific templates. Symbolic execution it provides assurance on the functionality and security of code and can also be used to generate high coverage test suites. Already, property-based techniques which extend symbolic execution have shown fantastic results for production security defects.

# References


[1] *W_06_00055_the_cost_of_defects*. [Online]. Available: https://www.experimentus.com/itm/W_06_00055_The_Cost_of_Defects.htm (visited on 04/17/2020).

[2] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," en, *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, Jan. 2019, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2017.2755013. [Online]. Available: https://ieeexplore.ieee.org/document/8089448/.

[3] *Program-repair.org*. [Online]. Available: http://program-repair.org/ (visited on 04/17/2020).

[4] *Data Breaches Compromised 3.3 Billion Records in First Half of 2018**, en-US, Library Catalog: www.gemalto.com. [Online]. Available: https://www.gemalto.com/press/pages/data-breaches-compromised-3-3-billion-records-in-first-half-of-2018.aspx (visited on 04/17/2020).

[5] F. Gao, L. Wang, and X. Li, "BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, Singapore, Singapore: Association for Computing Machinery, Aug. 2016, pp. 786–791, ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970282. [Online]. Available: https://doi.org/10.1145/2970276.2970282.

[6] A. Abadi, R. Ettinger, Y. A. Feldman, and M. Shomrat, "Automatically fixing security vulnerabilities in Java code," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. OOPSLA '11, Portland, Oregon, USA: Association for Computing Machinery, Oct. 2011, pp. 3–4, ISBN: 978-1-4503-0942-4. DOI: 10.1145/2048147.2048149. [Online]. Available: https://doi.org/10.1145/2048147.2048149.

[7] S. Ma, D. Lo, T. Li, and R. H. Deng, "CDRep: Automatic Repair of Cryptographic Misuses in Android Applications," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16, Xi'an, China: Association for Computing Machinery, May 2016, pp. 711–722, ISBN: 978-1-4503-4233-9. DOI: 10.1145/2897845.2897896. [Online]. Available: https://doi.org/10.1145/2897845.2897896.

[8] MainPage, *CWE - Common Weakness Enumeration*. [Online]. Available: https://cwe.mitre.org/ (visited on 04/18/2020).

[9] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury, "A correlation study between automated program repair and test-suite metrics," en, in *Proceedings of the 40th International Conference on Software Engineering*, Gothenburg Sweden: ACM, May 2018, pp. 24–24, ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3182517. [Online]. Available: https://dl.acm.org/doi/10.1145/3180155.3182517.

[10] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response," in *2016 IEEE Symposium on Security and Privacy (SP)*, ISSN: 2375-1207, May 2016, pp. 618–635. DOI: 10.1109/SP.2016.43.

[11] J. Jiang, Y. Xiong, and X. Xia, "A manual inspection of Defects4J bugs and its implications for automatic program repair," en, *Science China Information Sciences*, vol. 62, no. 10, p. 200 102, Oct. 2019, ISSN: 1674-733X, 1869-1919. DOI: 10.1007/s11432-018-1465-6. [Online]. Available: http://link.springer.com/10.1007/s11432-018-1465-6.

[12] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, Baltimore, MD, USA: Association for Computing Machinery, Jul. 2015, pp. 24–36, ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771791. [Online]. Available: https://doi.org/10.1145/2771783.2771791.

[13] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," en, *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, Aug. 2017, ISSN: 1573-7616. DOI: 10.1007/s10664-016-9470-4. [Online]. Available: https://doi.org/10.1007/s10664-016-9470-4.

[14] P. Agarwal and A. P. Agrawal, *Fault-localization techniques for software systems: A literature review*, Sep. 2014. [Online]. Available: https://doi.org/10.1145/2659118.2659125.

[15] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, ISSN: 1941-0026, Jun. 2008, pp. 162–168. DOI: 10.1109/CEC.2008.4630793.

[16] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan. 2012, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.2011.104.

[17] M. Martinez and M. Monperrus, "ASTOR: A program repair library for Java (demo)," en, in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, Saarbrücken, Germany: ACM Press, 2016, pp. 441–444, ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2948705. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2931037.2948705.

[18] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, ser. GECCO '11, Dublin, Ireland: Association for Computing Machinery, Jul. 2011, pp. 1427–1434, ISBN: 978-1-4503-0557-0. DOI: 10.1145/2001576.2001768. [Online]. Available: https://doi.org/10.1145/2001576.2001768.

[19] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, Hong Kong, China: Association for Computing Machinery, Nov. 2014, pp. 306–317, ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635898. [Online]. Available: https://doi.org/10.1145/2635868.2635898.

[20] M. Martinez and M. Monperrus, "Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor," en, in *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 65–86, ISBN: 978-3-319-99241-9. DOI: 10.1007/978-3-319-99241-9_3.

[21] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2Fix: Automatically Generating Bug Fixes from Bug Reports," in *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*, ISSN: 2159-4848, Mar. 2013, pp. 282–291. DOI: 10.1109/ICST.2013.24.

[22] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, May 2013, pp. 802–811. DOI: 10.1109/ICSE.2013.6606626.

[23] J. S. Bradbury and K. Jalbert, "Automatic Repair of Concurrency Bugs," en, p. 2,

[24] Yiming Xiang, *Automatic patch generation for security vulnerability*. MSc Thesis, University of Southampton, Sep. 2019.

[25] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, May 2013, pp. 772–781. DOI: 10.1109/ICSE.2013.6606623.

[26] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using Safety Properties to Generate Vulnerability Patches," en, in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2019, pp. 539–554, ISBN: 978-1-5386-6660-9. DOI: 10.1109/SP.2019.00071. [Online]. Available: https://ieeexplore.ieee.org/document/8835226/.

[27] Mingyue Yang, "Using Machine Learning to Detect Software Vulnerabilities," PhD thesis, University of Toronto. [Online]. Available: https://security.csl.toronto.edu/wp-content/uploads/2020/01/shirley_yang_masc_thesis.pdf.

[28] M. Alkhalaf, A. Aydin, and T. Bultan, "Semantic differential repair for input validation and sanitization," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA: Association for Computing Machinery, Jul. 2014, pp. 225–236, ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610401. [Online]. Available: https://doi.org/10.1145/2610384.2610401.

[29] admin, *Dynamic Analysis vs. Static Analysis*, en, Library Catalog: software.intel.com. [Online]. Available: https://software.intel.com/en-us/inspector-user-guide-windows-dynamic-analysis-vs-static-analysis.

[30] K. Goseva-Popstajanova and A. Perhinschi, "Using Static Code Analysis Tools for Detection of Security Vulnerabilities," en, p. 35,

[31] *Semmle/ql*, original-date: 2018-07-31T16:35:51Z, Apr. 2020. [Online]. Available: https://github.com/Semmle/ql (visited on 04/17/2020).

[32] F. Y. Assiri and J. M. Bieman, "An Assessment of the Quality of Automated Program Operator Repair," in *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, ISSN: 2159-4848, Mar. 2014, pp. 273–282. DOI: 10.1109/ICST.2014.40.

[33] X. Kong, L. Zhang, W. E. Wong, and B. Li, "Experience report: How do techniques, programs, and tests impact automated program repair?" In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2015, pp. 194–204. DOI: 10.1109/ISSRE.2015.7381813.

[34] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2017, pp. 637–647. DOI: 10.1109/ASE.2017.8115674.

[35] *SerVal-DTF/FL-VS-APR*, en, Library Catalog: github.com. [Online]. Available: https://github.com/SerVal-DTF/FL-VS-APR (visited on 04/27/2020).

[36] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Amsterdam, Netherlands: Association for Computing Machinery, Jul. 2018, pp. 298–309, ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213871. [Online]. Available: https://doi.org/10.1145/3213846.3213871.

[37] *GitHub Surveys Open Source: Documentation, License, Usage at Work*, Library Catalog: www.infoq.com. [Online]. Available: https://www.infoq.com/news/2017/06/github-survey-open-source/ (visited on 05/26/2020).

[38] V. Debroy and W. E. Wong, "Using Mutation to Automatically Suggest Fixes for Faulty Programs," in *Verification and Validation 2010 Third International Conference on Software Testing*, ISSN: 2159-4848, Apr. 2010, pp. 65–74. DOI: 10.1109/ICST.2010.66.

[39] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk, "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, ISSN: 1534-5351, Feb. 2019, pp. 479–490. DOI: 10.1109/SANER.2019.8668043.

[40] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," en, p. 4,

[41] P. E. Black, "Juliet 1.3 test suite: Changes from 1.2," en, National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST TN 1995, Jun. 2018, NIST TN 1995. DOI: 10.6028/NIST.TN.1995. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/TechnicalNotes/NIST.TN.1995.pdf (visited on 04/17/2020).

[42] N. Smith, F. Tomassetti, and D. van Bruggen, *JavaParser: Visited*, en.

[43] *1N3/IntruderPayloads*, en, Library Catalog: github.com. [Online]. Available: https://github.com/1N3/IntruderPayloads.

[44] *Back to Agile's basics with "Uncle Bob" Martin (The Changelog #367)*, en, Library Catalog: changelog.com. [Online]. Available: https://changelog.com/podcast/367 (visited on 05/26/2020).

[45] G. Candea and P. Godefroid, "Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances," en, in *Computing and Software Science: State of the Art and Perspectives*, ser. Lecture Notes in Computer Science, B. Steffen and G. Woeginger, Eds., Cham: Springer International Publishing, 2019, pp. 505–531, ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_24. [Online]. Available: https://doi.org/10.1007/978-3-319-91908-9_24.

# Appendix A: Progress Report Documents

| SKILLS | RATING (1-5) |
|---|---|
| Technical Skills | |
| Python | 1 |
| Java | 3 |
| Design | 3 |
| Testing | 2 |
| Analysis Skills | |
| Writing | 5 |
| Researching | 4 |
| Evaluation | 3 |

Figure 30: Initial skills audit from progress report

| PROBLEM | RISK | PLAN |
|---|---|---|
| Laptop may be damaged or lost | 1*4 (4) | Code will be version managed on GitHub. If Laptop is unusable then use machines in the labs. |
| Difficulty developing Java Test Suites | 2*4 (8) | Guidance will be sought from peers and other lecturers at the University. The prototyping stage of development should make this process easier. |
| Final Iteration of project doesn't succeed as hoped | 3*2 (6) | Strict Documentation will be kept ensuring the development of the project is shown. Strong evaluation and reflection will be given to help understand the problem for future research |
| Lack of Code Repositories for testing patches | 2*4 (8) | Speak with supervisor or previous years students who have worked on similar projects. |
| Development Problems with Code Repositories | 2*3 (6) | Attempt to install Open Source Repositories with fresh Virtual Machines to ensure the greatest chance of success or use lab machines. |
| Problems licensing or using COTS Patch Generation tools | 4*3 (12) | Speak with supervisor about this, may be a non-issue but it is very important to reduce this risk. Educational licenses may be available. |

Figure 31: Contingency plan from progress report

# Appendix B: Program Output

Truncated reports have been included below from our 4 defective programs in SECdefects.

```
package cc14g17.SECdefects;

import ...

/**
 * AUTOMATICALLY GENERATED TEST SUITE
 * NOTE: TO COMPILE CORRECTLY PLEASE REMOVE { } AROUND INITIALIZATION FIELD AT TOP
 @author cc14g17
 */
public class CWE20_Improper_Input_ValidationAUTOGEN_EXPLOIT_Test {

    private CWE20_Improper_Input_Validation cwe20;

    @Before
    public void setUp() { cwe20 = new CWE20_Improper_Input_Validation(); }

    @Test
    public void badWithdraw1() {
        cwe20.badWithdraw( amount: 0);
        Assert.assertEquals( expected: 0,cwe20.getBalance(), delta: 0.001);
    }

    @Test
    public void badWithdraw2() {
        cwe20.badWithdraw( amount: 1);
        Assert.assertEquals( expected: -1,cwe20.getBalance(), delta: 0.001);
    }

    @Test
    public void badWithdraw3() {
        cwe20.badWithdraw( amount: 2);
        Assert.assertEquals( expected: -2,cwe20.getBalance(), delta: 0.001);
    }

    @Test
    public void badWithdraw4() {
        cwe20.badWithdraw( amount: 3);
```

Figure 32: Testing screenshot from CWE-20 test suite generation


Figure 33: Testing screenshot from CWE-22 test suite generation

```java
package cc14g17.SECdefects;

import ...

/**
 * AUTOMATICALLY GENERATED TEST SUITE
 * NOTE: TO COMPILE CORRECTLY PLEASE REMOVE { } AROUND INITIALIZATION FIELD AT TOP
 @author cc14g17
 */
public class CWE89_SQL_InjectionAUTOGEN_EXPLOIT_Test {

    private CWE89_SQL_Injection cwe89;

    @Before
    public void setUp() { cwe89 = new CWE89_SQL_Injection(); }

    @Test
    public void badLogin1() {
        cwe89.badLogin( username: "OR 1=1", password: "OR 1=1");
        Assert.assertFalse(cwe89.isLoggedIn());
    }

    @Test
    public void badLogin2() {
        cwe89.badLogin( username: "OR 1=0", password: "OR 1=0");
        Assert.assertFalse(cwe89.isLoggedIn());
    }

    @Test
    public void badLogin3() {
        cwe89.badLogin( username: "OR x=x", password: "OR x=x");
        Assert.assertFalse(cwe89.isLoggedIn());
    }

    @Test
    public void badLogin4() {
        cwe89.badLogin( username: "OR x=y", password: "OR x=y");
```

Figure 34: Testing screenshot from CWE-89 test suite generation

```java
package cc14g17.SECdefects;

import ...

/**
 * AUTOMATICALLY GENERATED TEST SUITE
 * NOTE: TO COMPILE CORRECTLY PLEASE REMOVE { } AROUND INITIALIZATION FIELD AT TOP
 * @author cc14g17
 */
public class CWE125_Out_of_Bounds_ReadAUTOGEN_EXPLOIT_Test {

    private CWE125_Out_of_Bounds_Read cwe125;

    @Before
    public void setUp() { cwe125 = new CWE125_Out_of_Bounds_Read(); }

    @Test
    public void badGetValue1() { cwe125.badGetValue( index: 0); }

    @Test
    public void badGetValue2() { cwe125.badGetValue( index: 1); }

    @Test
    public void badGetValue3() { cwe125.badGetValue( index: 2); }

    @Test
    public void badGetValue4() { cwe125.badGetValue( index: 3); }

    @Test
    public void badGetValue5() { cwe125.badGetValue( index: 4); }

    @Test
    public void badGetValue6() { cwe125.badGetValue( index: 5); }

    @Test
    public void badGetValue7() { cwe125.badGetValue( index: 6); }
```

Figure 35: Testing screenshot from CWE-125 test suite generation

# Appendix C: Additional Analysis

## FindBugs Report

Produced using FindBugs3.0.1.

Project: SECdefects[SECdefects]

### Metrics

289 lines of code analyzed, in 7 classes, in 1 packages.

| Metric | Total | Density* |
|---|---|---|
| High Priority Warnings | 1 | 3.46 |
| Medium Priority Warnings | | NaN |
| **Total Warnings** | **1** | **3.46** |

*(* Defects per Thousand lines of non-commenting source statements)*

### Summary

| Warning Type | Number |
|---|---|
| Security Warnings | 1 |
| **Total** | **1** |

## Warnings

Click on each warning link to see a full description of the issue, and details of how to resolve it.

### Security Warnings

| Warning | Priority | Details |
|---|---|---|
| Nonconstant string passed to execute or addBatch method on an SQL statement | High | cc14g17.CWE89_SQL_Injection.badLogin(String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement<br><br>In file CWE89_SQL_Injection.java, line 51<br>In class cc14g17.CWE89_SQL_Injection<br>In method cc14g17.CWE89_SQL_Injection.badLogin(String, String)<br>At CWE89_SQL_Injection.java:[line 51]<br>At CWE89_SQL_Injection.java:[line 51] |

## Warning Types

### Nonconstant string passed to execute or addBatch method on an SQL statement

The method invokes the execute or addBatch method on an SQL statement with a String that seems to be dynamically generated. Consider using a prepared statement instead. It is more efficient and less vulnerable to SQL injection attacks.

Figure 36: Full Results of findbugs static analysis

# Appendix D: Original Project Brief

Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

**Callum Connolly**

November 14, 2019

---

# Improving Automatic Software Repair for Security Vulnerabilities by Automatically Generating Test Suites

---

*Project supervisor*:
Dr Julian Rathke

*Project Description*:
Automatic Program Repair relies greatly on the using the passing and failing cases of test suites to guide the localization and fixing of defects. So far development in this area, even including what makes a good test suite, is fairly limited relying on Hand made test suites.

The project will have an emphasis on security vulnerabilities, taking in a program and outputting a series of test cases, ideally for both to be put into guide repair tools to see if there is an improvement to fixes. Program input will be gathered with existing defective code repositories, as well as artificially defective programs with synthetic bugs.

A project brief submitted for the award of
BSc Computer Science