MASTER EN SCIENCES INFORMATIQUES

Towards Security Aware Mutation Testing

Loise, Thomas

Award date:
2017

Awarding institution:
Universite de Namur

Link to publication

Université de Namur
Faculty of Computer Science
Academic Year 2016–2017

Towards Security Aware Mutation Testing

Thomas Loise

Internship mentor:	Mike Papadakis
Supervisor:	________________________ (Signed for Release Approval - Study Rules art. 40) Patrick Heymans
Co-supervisor:	Gilles Perrouin & Xavier Devroey

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université of Namur

ABSTRACT

LOISE, THOMAS. Towards Security-aware Mutation Testing. (Under the direction of Patrick Heymans.)

Mutation analysis forms a popular software analysis technique that has been demonstrated to be useful in supporting multiple software engineering activities. Yet, the use of mutation analysis in tackling security issues has received little attention. In view of this, we design security aware mutation operators to support mutation analysis. Using a known set of common security vulnerability patterns, we introduce 15 security-aware mutation operators for Java. We then implement them in the PIT mutation engine and evaluate them. Our preliminary results demonstrate that standard PIT operators are unlikely to introduce vulnerabilities similar to ours. We also show that our security-aware mutation operators are indeed applicable and prevalent on open source projects, providing evidence that mutation analysis can support security testing activities.

L'analyse de mutation est une technique d'analyse de logiciel qui a déjà démontré son utilité dans plusieurs activités d'ingénierie du logiciel. Cependant, l'analyse de mutation n'a été que peu utilisée dans le but de faire face aux problèmes qui relèvent de la cyber-sécurité. Dans cette optique, nous concevons des opérateurs de mutation sensibles à la cyber-sécurité, qui viendront supporter l'analyse de mutation. En utilisant un ensemble connu de motifs de vulnérabilités, nous présentons 15 opérateurs sensibles à la cyber-sécurité pour le langage de programmation Java. Ensuite, nous les implémentons dans le moteur de mutation PIT et nous les évaluons. Nos résultats préliminaires démontrent qu'il est peu probable que les opérateurs standards de PIT introduisent des vulnérabilités similaires aux nôtres. De plus, nous montrons que nos opérateurs cyber-sensibles sont effectivement utilisables dans des projets open-source, ce qui prouve que l'analyse de mutation peut supporter des activités de testing pour la cyber-sécurité.

ACKNOWLEDGEMENTS

This internship gave me the opportunity to discover a very large panel of technical skills as security testing, web application building or byte-code analysis but it also gave me the occasion to develop some social skills that are inherent to the academic environment. However, this was most of all the occasion to work with great people.

First, I would like to thank Professor Yves Le Traon and Professor Patrick Heymans for the opportunity they gave me to perform this internship and for their great support towards future directions of this work. I would also like to express my sincere gratitude to my supervisors and internship mentor Gilles Perrouin, Xavier Devroey and Mike Papadakis for their energy and patience during and after the internship. Moreover, I would like to thank you all for the opportunity you gave me to attend and present our work at an international workshop (Mutation2017) and meet the experts of the mutation testing domain.

I am also very grateful to those who gave me support in more subtle ways and also where the most welcoming people I have ever met, members of the Serval Team, Ludovic Mouline, Marinos Kintis, Thierry Titcheu, Matthieu Jimenez and Rahul Kumar Dutta. I hope to see you again soon.

My profound appreciation goes to Anestis Taskmalis, a member of the the research center, without who all of this would certainly not have been possible. Thank you for your friendship and unconditional support.

Last but not least, I would like to thank my family for their patience at home and especially my mother for her help in the redaction of this document.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Context

In our world, a humongous amount of activities rely on IT. Hence, web-based services support banks, web commerce, health organizations, and even governments activities. These web applications[1] are supposed to be trustworthy. Hence, we cannot imagine to use a webapp for an economical purpose without trust. However, studies and past exploits have demonstrated that webapps are also prone to security issues. Therefore, security testing has been integrated in the software development cycle, translating itself in various activities, specifically designed to insure a given level of security for the application under development.

However, security can be viewed as a common non-functional requirement, and testing techniques may be adapted in order to cover those requirements. In this context, we explore the possibility of adapting mutation testing [9, 10], a popular fault-based testing technique, for security requirements and at code level. As every fault-based technique, mutation testing provides guarantees that the software under analysis is free from specific types of faults [11]. Also, the technique is known to form a flexible and effective way to perform testing, this last fact explaining its popularity. Among others, mutation testing is used to guide test generation [12], to perform test assessment [13] and to uncover subtle faults [14]. It works by generating syntactically altered program versions of the system under test, called *mutants*. These alterations substitute programmers faults [15] and therefore can be used to determine the adequacy of test suites at detecting those faults. The method's flexibility relies on the possibility of tuning the introduced alterations [10] for specific kind of faults. Thus, mutation testing can be used for almost every purpose testing aims at.

[1]We will use the equivalent term "webapps" for the remaining of the document.

This last fact enables us to use mutation testing for security, as we may design alterations specifically designed to introduce vulnerabilities. By taking advantage of the fault-based nature of the technique, mutation testing may ensure that security-aware faults are not present in the core of the system under test and, through regression tests, that these will not appear in its future evolutions.

1.2 Research problem

There is a large diversity of different mutation operators sets, each specifically designed to guide the tester in a precise context. Hence, among others, there are mutation operators for OO-program testing [16], SQL database queries testing [17] and integration testing [18]. However, to the best of our knowledge, there is no mutation operator set specifically designed for security testing at code level. Therefore, we aim at addressing this research problem: May we design security-aware mutation operators that are helpful in security testing for web applications? As many webapps are implemented in JAVA, we focus on this programming language in our research. In order to answer the problem, we will need to investigate to which extent traditional mutation operators for JAVA are already suitable for security testing.

1.3 Overview of the approach

To address the research problem, we designed ourselves security-aware mutation operators, *i.e*, mutation operators specifically designed to introduce security bugs in web applications. Therefore, we took inspiration on common security bug patterns encoded by a well-known static analysis tool called FindBugs-sec-plugin[2] [19]. The bug patterns used by the static analysis tools aim at identifying potential issues with the code under analysis. Therefore, they point-out the presence of potential bugs and not opportunities for injecting them as it is done by the mutation operators. To cover this last point and design our security-aware mutations, we manually analyzed the bug patterns, inferred the classes of faults they represent and inverted them, *i.e.*, we defined rules that introduce these defects. We have implemented our mutations in PIT, a popular mutation testing tool [3] and provide initial exploratory results showing their applicability and difference from the traditional mutation operators. Thus, we applied both the traditional and our operators on four subject programs and validated the presence of potential vulnerabilities using FindBugs. Our results demonstrate that traditional operators are ineffective in introducing such security-aware faults.

[2]Find Security Bugs is a plugin for FindBugs that aims at identifying security issues in JAVA webapps.

Overall, our security related faults represent simple vulnerabilities, which can form an initial step for defining security-aware testing requirements. We believe that these requirements are particularly useful when building regression test suites of webapps. Furthermore, our operators can be particularly useful in evaluating and comparing fuzzing or other security testing tools. In summary, our research makes the following contributions:

1. We design 15 security-aware mutation operators for supporting security mutation testing.
2. We extend PIT so that it applies both traditional and security-aware mutation testing. To support future research, we make our implementation publicly available.
3. We make an initial assessment of our operators demonstrating their prevalence and the potential weaknesses of the traditional operators using large real-world projects.

The major outcomes of the research exposed in this thesis were published [20] in *Mutation2017*, the 12th in the series of international workshops focusing on mutation analysis collocated with the international conference on software testing verification and validation that was held in mid April in Tokyo this year. I had the opportunity to present these results at the workshop. The published paper is attached to the thesis in the appendix section.

1.4 Thesis structure

The thesis is organized as follows: Chapter 2 provides the background and concepts needed for the understanding of the thesis. Chapter 3 exposes the state of the art in the research problem we try to address. Chapter 4 answers partly to the research problem by stating the mutation operators we propose. Chapter 5 details the research questions we formulate in order to answer formally the research problem risen. Then, the chapter gives details of the experimentation process and presents the case studies we analyzed. Chapter 5 also presents and discusses the results we obtained. Finally, chapter 6 gives a conclusion and proposes perspectives to the thesis.

Chapter 2

Background

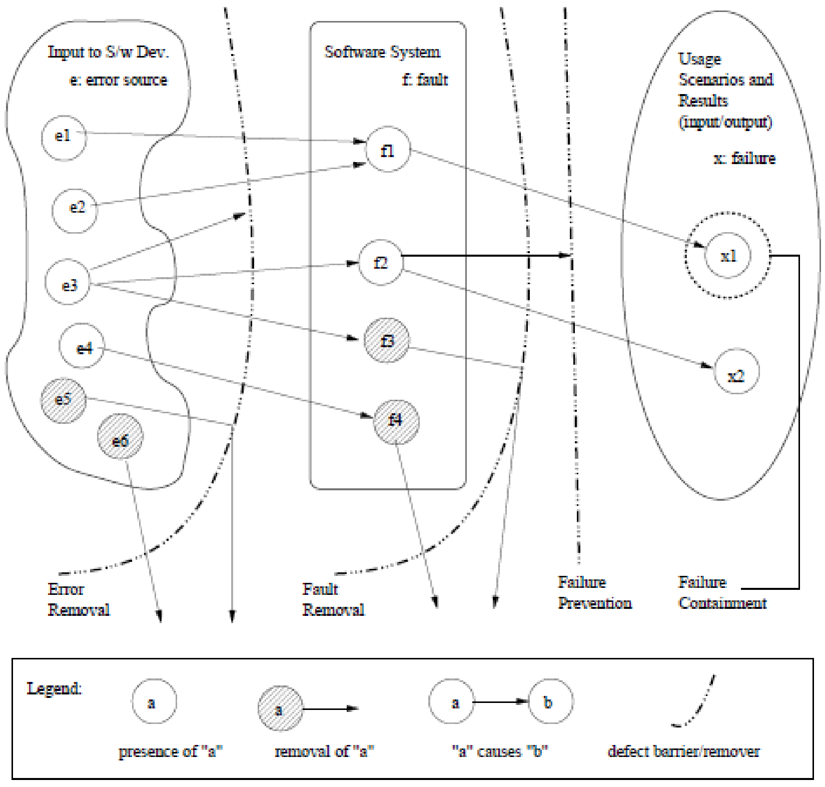
In this chapter, we present the concepts and theories needed for a good understanding of the outcomes of this study. We begin with an introduction to testing, then follow with an introduction to the mutation testing theory. We follow with the presentation of the mutation testing tool that we used during our experimentation, PITest. Then, we give main concepts of security testing also required for the understanding of the following chapters. Finally, we present a security static analyzer that we used during our experimentation, FindBugs.

2.1 Software Testing

Software engineering, as every human activity, is prone to errors. These errors can be introduced at any step of the software development life-cycle and can lead to software failures during the operation of the system. Figure 2.1 summarizes the relations between errors, faults and failures.Errors are made in the specification, design or coding activities. Those errors introduce faults in the system, that can result in failures. Failures are observable wrong behaviors of the software system. Faults are mistakes introduced in a software system.

Obviously, software should be free from faults. There are two ways to evaluate the correctness of software. One is *empirical verification*, which can guarantee the absence of faults using theorem proving or model-checking techniques. Unfortunately, it is unpractical and infeasible for large cases to apply these methods because of the complexity of their associated analyses. Thus, we recourse to software testing by exercising the software on finite domains of inputs/outputs. Though non-exhaustive, software testing can flexibility accommodate limited validation budgets.

Figure 2.1: The relation between errors, faults, and failures [1].



Undoubtedly, the goal we just described is very abstract. Indeed, there are many questions to be answered while designing tests: Which scenarios should I cover ? When do I have enough tests ? Therefore, this goal should be rephrased in a more concrete and understandable target: a coverage criterion.

A *coverage criterion* is a rule or a collection of rules guiding the test requirements design process. A *test requirement* is a specific element of a software artifact that a test case must cover or satisfy [10]. A *test suite*, a set of test cases, is therefore the product of satisfying several test requirements. The process of software testing can be summarized by the following activities.

1. Choose a coverage criterion.

2. Derive test requirements from the coverage criterion rules.

3. Satisfy the test requirements by implementing test cases, this giving a test suite.

Many coverage criteria exist. For instance, branch coverage's satisfaction requires to cover all possible decisions in a program, statement coverage's satisfaction requires to cover all the statements of the program under test. Criteria define the efforts to be provided in order to assess the partial absence of faults in the software. Moreover, some criteria are stronger than others, providing more complete fault test suites, by costs of efforts in the number of test cases to provide. Indeed, a criterion can be proven harder to satisfy than another, using the **subsuming relation**. Formally, given *C1* and *C2*, two coverage criteria, *C1* **subsumes** *C2* if and only if every test set that satisfies *C1* also satisfies *C2* [10].

Software testing is usually performed to assess the functional behavior of a system but it can also be used for testing non functional requirements such as security or the performance of the system under test[1]. Also, testing can be performed at different levels of granularity. One can test a complete system whereas it can also test each component or class individually. In every case, the key aspect of testing is the choice of a coverage criterion. Mutation testing, as any testing technique, presents one of these criteria. We note that mutation testing can be used at every level of granularity and for any requirement that can be stated for the system. We detail this technique in the next section.

[1]We will use the equivalent term "SUT" for the remaining of the document)

2.2 Mutation Testing

This section presents traditional mutation testing, its origins but also the concepts it is based on.

2.2.1 Traditional Mutation Testing

Mutation testing, like most testing technique, provides a coverage criterion. The *mutation criterion* is very powerful as it subsumes several other criteria [21].

Originally, mutation testing was proposed by DeMillo *et al.* Let us summarize the reasoning presented in [9, 10], the first proposition of mutation testing from DeMillo *et al.*

First, the authors define *the coupling effect hypothesis.* It is the hypothesis that, should a test suite be capable to distinguish simple syntactical changes in a program, it would be able to distinguish complex changes. The rationale behind this is that complex changes are composed of simple ones. DeMillo *et al.* define faults as syntax differences between the working program and the program the programmer intended to realize. Also, they state that if a test suite is capable of distinguishing simple differences in the program, it should be capable to distinguishing all the complex differences subsumed by these slight differences. The recommendation arising from the coupling effect is to evaluate test suites using their capability of detecting slight changes (faults) in a program, as the complex changes (complex faults) should also be unveiled by such test suites.

The second defined principle is *the competent programmer hypothesis.* It states that programs created by programmers are very close to the program the programmers intend to create. The authors do **not** validate this hypothesis, rather accept it as an empirical principle. The competent programmer hypothesis states that, by modifying a program using simple changes rules, one can simulate bug/fault introduction. This final principle was formalized by Geist *et al.* [22] : "If the software contains a fault, it is likely that there is a mutant that can only be killed by a test case that also reveals the fault".

Finally, relying on these hypotheses, a coverage criterion is defined. To be satisfied, the mutation testing criterion requires the test suite to be capable of distinguishing a set of artificially introduced faults in the original program (*i.e* mutants). The mutation testing process, based on the mutation testing criterion is described in the figure 2.2. We comment the figure bellow.

Given an original program P and a test suite T,

1. Introduce faults in the original program P, creating mutants of the program under test (P'). The program under test is declined in several mutants containing one or several faults. These faults are introduced following fault-introducing rules called *mutation operators.*

2. Run T on P. If the test suite is already capable of identifying real faults in the program, the program should be corrected. Then, return to (1). If not, assess the quality of the test suite by verifying that the test suite is capable of finding the introduced faults[2]. A mutant distinguised from the original program is referred to as *killed* whereas the opposite behavior is referred to as *lived.* If a mutant LM is lived, the test suite T is incapable of distinguishing it from P, meaning that T is unable to identify an introduced fault. The cause can either be that

   - T is incomplete, it should be completed with more test cases until it is capable of killing the mutant LM.
   - LM is actually synthetically different from the original program, but semantically equivalent. Therefore, the tester should ignore this mutant. In traditional mutation testing, this verification is performed manually, introducing a time consuming task. In the process described in figure 2.2, those *equivalent mutants* are ignored after being uncovered.

3. Correct/Improve the test suite and replay. If this test suite is not capable of discovering all the introduced faults, the tester should either create new test cases or modify existing ones. Then, he should replay the process and return to the first step (1).

The mutation testing criterion can be summarized by trying to maximize the mutation score, defined as

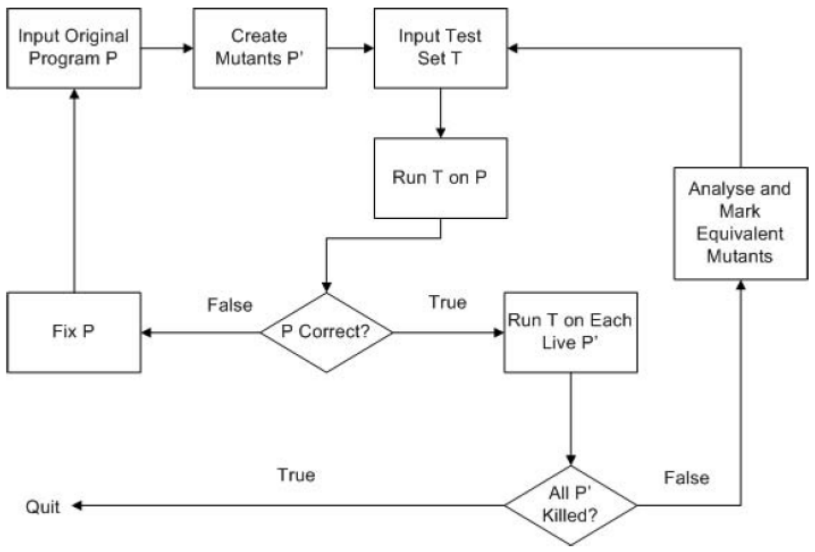
$$MS = \frac{\textit{mutants killed}}{\textit{mutants generated - equivalent mutants}}$$

.

Mutation testing has its drawbacks, too. First, it is **expensive for practical use** [2]. This cost is mainly due to the number of mutants to be generated and compiled, the number of equivalent ones to be assessed manually, the test execution time and to the lack of efficient mutation tools. Also, it **strongly relies** on the mutation operators and on the tool used by the tester [23].

[2]A tool is very useful in this process by highlighting to the tester the introduced faults that were not uncovered by his test suite.

Figure 2.2: The traditional mutation testing process [2].



On the other hand, although the mutation testing process is time consuming, the method forms one of the most efficient coverage criterion available for testing [24] [25] [26]. Also, the technique is known to be flexible, as one can use specific mutation operators, corresponding to the kind of faults he wants to assess his program is free from. Keeping this fact in mind, we investigate whether mutation can be used for security testing, using security-aware mutation operators. We explain the design process of these operators in chapter 4. We clarify in the next section our choice of PITest as an ideal mutation tool in order to implement and experiment our mutation operators.

2.2.2 Mutation testing tool : the choice of PITest

In order to evaluate our mutation operators, we needed to implement them in a mutation testing tool. Many of these tools were provided throughout the research domain's exploration. For instance, considering only JAVA programs mutation, more than 8 mutation testing tools developed both by academics and open-source enthusiasts could be cited [27].

In particular, we considered to use *MuJava*, *Major* and *PIT*, as these tools are the most widely used in the state of the art [23].

- MuJava [28] was developed in a collaboration between *Korea Advanced Institute of Science and Technology (KAIST)* and *George Mason University, USA* [28]. It is one of the oldest JAVA mutation testing tools, as it was introduced to the academics in 2006. The tool works by mutating the source code, but avoids the compilation of all the mutants using the mutant schemata approach. Hence, it constructs the representation of the mutants by adding conditional statements to the program under test, triggering (or not) a mutation. Hence, every of these conditional statements depend on a different `Boolean` that is added to the inputs of the program under test. The only way to describe new mutants in MuJava is to manipulate its sources.

- Major (Mutation Analysis in a JAVA cOmpileR) [29] is more recent than MuJava (2011). It is a mutation testing framework, integrated into the JAVA compiler (enabled with a compiler option). The generation of mutants manipulates the Abstract Syntax Tree of the program under test. Major relies on the compiler for the original program parsing, avoiding a main challenge of mutation testing tools: the version compliance. The test running phase is handled by a mutation analyzer, build on top of Apache Ant. Major provides a *DSL* to describe new mutation operators.

- PIT [3] is a mutation testing tool mainly developed by open-source mutation testing enthusiasts. However, it has also been used a lot in research. The tool performs several optimizations, like test prioritization (also present in Major), but its main characteristic is that it performs mutation on the program's byte code in order to avoid the compilation of the mutants. To keep track of the mutations at source code level, it utilizes a run-time mapping between byte and source code and prints its results in a html (or csv) report (see figure 2.3). PIT is easily extendible, describing a new mutation operator is achievable by implementing a standard visitor pattern class. This last statement forms one of the reasons for which we chose PIT.

We mainly took the tool's popularity as a criterion of choice and chose *PIT*. Indeed, it has a strong community, exchanging on a Google group[3] (more than 400 subjects) and a *GitHub* issues tracker[4] (more than 93 issues) since 2011. It also provides support for several very important development tools, *e.g* an *eclipse* plugin, a *maven* plugin and an *Ant* custom task that are all maintained by enthusiasts of the domain. Therefore, we strongly believe that PIT will thrive in the future and engage the impact we seek for the development of the idea we introduce.

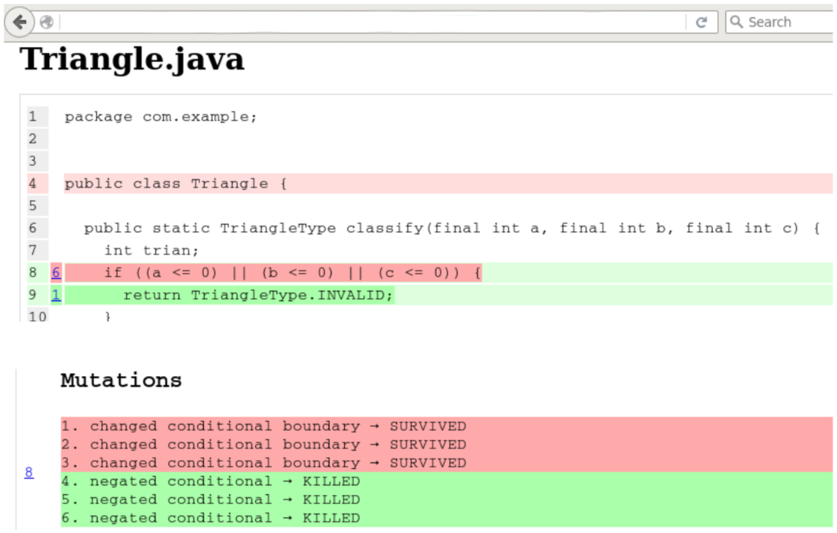
Also, we could note the relative easiness of describing new mutation operators in PIT, as it relies on an intermediate level manipulation API, the ASM[5] JAVA byte code manipulation

[3]The Google group of PIT's users is available at https://groups.google.com/forum/#!forum/pitusers

[4]The GitHub issue tracker of PIT is available at https://github.com/hcoles/pitest/issues

[5]ASM does not stand for anything. It is a reference to the keyword in C which allows some functions to be implemented in assembly language

Figure 2.3: Example of html report of PIT. Adapted from [3].



Triangle.java

```
1   package com.example;
2
3
4   public class Triangle {
5
6     public static TriangleType classify(final int a, final int b, final int c) {
7       int trian;
8 6     if ((a <= 0) || (b <= 0) || (c <= 0)) {
9 1       return TriangleType.INVALID;
10      }
```

Mutations

```
8  1. changed conditional boundary → SURVIVED
   2. changed conditional boundary → SURVIVED
   3. changed conditional boundary → SURVIVED
   4. negated conditional → KILLED
   5. negated conditional → KILLED
   6. negated conditional → KILLED
```

framework[6]. More details about the way to implement new mutation operators in PIT are available in chapter 4.

Moreover, none of the tools we presented is known to subsume the others, meaning that building a test suite with one will kill all the mutants generated by the other. Furthermore, their effectiveness (their average capability at killing each others mutants) is comparable [23]. Therefore, no choice could rely on the “best tool to use”. Finally, although PIT’s effectiveness was below the two others, the tendency could change soon, as researchers developed new operators to improve PIT’s results [30].

2.3 Software security

Security is becoming a key aspect in software engineering mainly because of the costs uncovered by a successful attack. Hence, attackers may outbreak down-times to the system, modify its data, or simply get access to critical information. Whereas critical operations such as banking, government data or healthcare are now managed by Information Systems, a single attack leading to one of the discussed leaks may be dramatic in terms of reputation and currency costs for a company but may also result in dramatic events for people [31].

[6]The ASM JAVA byte code manipulation framework is detailed at http://asm.ow2.org/, the website provides several tutorials and a user guide. In this study, we used the fourth version of the library.

Of course, a system may be protected by the use of firewalls, intrusion detection systems or anti-virus, providing a cost efficient response but the experience has shown that building secure systems is still needed because those barriers may be bypassed.

Software security is a software's ability to resist to attacks, its ability to guarantee a functional behavior when attacked [32].

Tian-yang*et al.* [33] define software security testing as the process of verifying the software's security features' consistency with its design. The idea is to verify that the security features are implemented and are providing an efficient protection. Hence, the authors divide security testing into two goals.

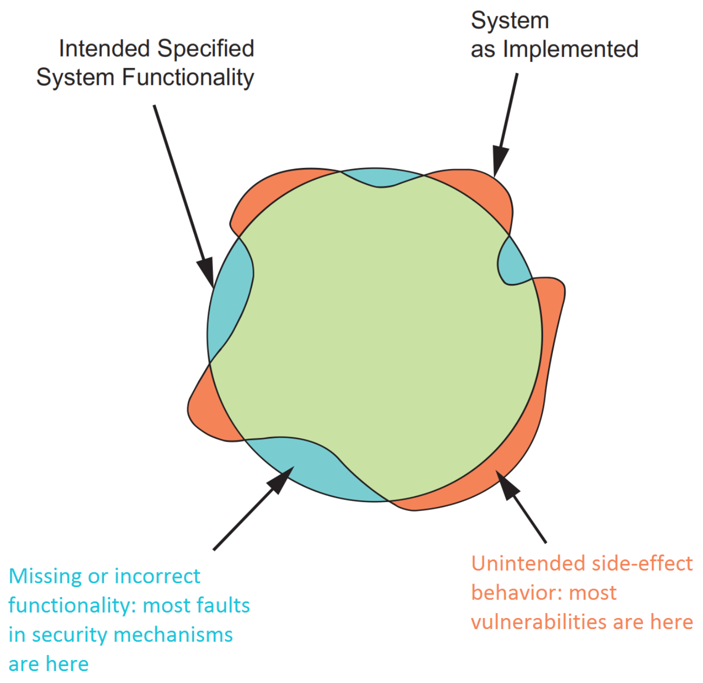
The first is called *Security functional testing.* It relies on the verification and validation of the implementation of the security requirements, meaning fault-testing applied to the system's security requirements. The second is *Security vulnerability testing*, the verification that there are no vulnerability left in the system. A *vulnerability* is referred to as a flaw, relying in the system's design or implementation, that may be exploited by attackers. The first goal can be seen as a developer's vision of security testing, the second is an attacker's.

The first goal is also referred to as positive security testing [32], the verification that security properties are satisfied for a number of assets in the system. The second is also referred to as negative or destructive security testing. Positive security testing will verify the security features of a system using legal/intended inputs. Negative security testing will on the other hand test the system with non-intended inputs. Therefore, the first technique can rely on traditional fault testing whereas the second will require a more specific expertise from the tester. A last way to describe the two goals is to perceive the first as the verification that the system does what it is supposed to do, and the second is the verification that the system never does what it should not do, referring in both cases to its security.

The complementarity of the two approaches is demonstrated in the following figure 2.4. Hence, the first approach will stretch green area to fit to the circle and the second will prevent the green area to exceed from this circle.

An overview of means of including security in the software development life-cycle is developed in the next section.

Figure 2.4: Representation of a System's security. The circle represents its specification. The system as implemented is represented by the green and the orange parts. "Most faults in security mechanisms are related to missing or incorrect functionality, most vulnerabilities are related to unintended side-effect behavior". Adapted from [4].


Intended Specified System Functionality
System as Implemented
Missing or incorrect functionality: most faults in security mechanisms are here
Unintended side-effect behavior: most vulnerabilities are here

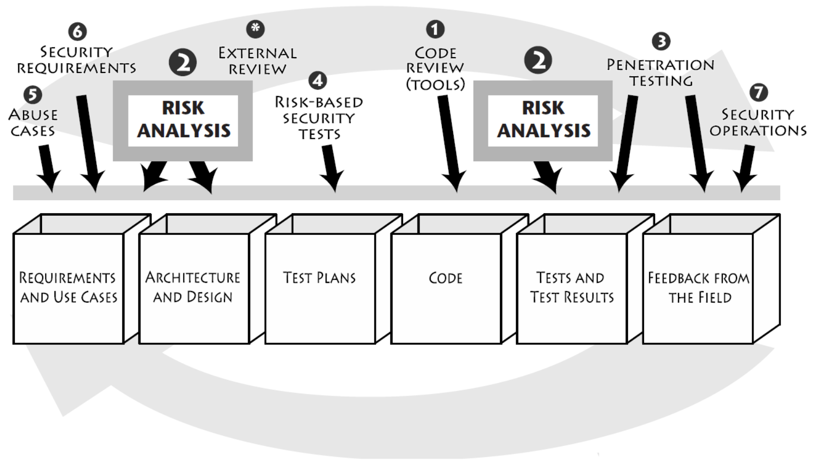
2.3.1 Security in the software development life-cycle

Security concerns the whole system under construction. Usually, several barriers are implemented in order to protect a system. These several protections should be designed in advance in order to be additive/synergistic so that each protection secures a part of the system while minimizing the breaches in between each protection.

Therefore, security, as any other non-functional requirement, must be included in the software development life-cycle. The overview of the practices that should be applied in order to secure a software is developed following McGraw [5] on figure 2.5. These practices are security activities taking place in the development life-cycle. Further explanations about the proposed techniques are detailed in section 2.3.2.

We can see that every phase of the software development cycle has several security building activities. Usually, one considers securing its system to a certain point, using risk analysis [4]. *Security requirements* cover both positive and negative security views. *Abuse cases* may repre-

Figure 2.5: The software development life-cycle, aggregated with security practices applicable at each of its steps [5].



sent the requirements for negative security. Abuse cases describe how the system reacts when it is under attack. At every level of design, practitioners should use risk analysis. *Risk analysis* is the process of ranking possible attacks on the system by their probability of appearance but also by their cost. The design must be thought in order to mitigate and/or suppress risks on the system. Then, it should be analyzed again to document possible attacks. *External review*, performed by external experts, may also be used. At code level, static analysis and dynamic security testing should be performed. Static analysis focuses on the discovery of known vulnerabilities. Security testing must, as explained before, cover both positive and negative requirements. Then, prior to delivery, *penetration testing* may also be performed. Because penetration testing may potentially be infinite, it should be driven via risk assessment. Penetration testing is very valuable as it tests the system within its environment. If a vulnerability cannot be secured, monitoring the systems' behavior against attacks relying on this vulnerability is also recommended. This step of mitigating impacts of those unprotectable attacks is called security operations. Because it uses the information's gained throughout the previous activities, this phase is performed at the end.

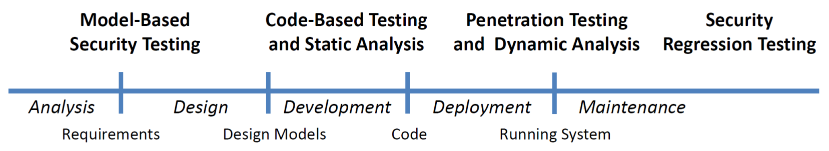
In the next section, several security testing techniques ensuring this non-functional requirement at every phase of the development life-cycle are developed.

2.3.2 Security testing in the software development life-cycle

Many security testing techniques exist. As our goal is not to provide an exhaustive enumeration of these techniques, we rather give a summary of the techniques that are applicable at every phase of the development life-cycle that we could discover in [4], and that were originally proposed in OWASP's[7] testing guide [34].

The figure 2.6 gives the families of techniques proposed by the survey and their moment of appliance in the development life-cycle.

Figure 2.6: The software development life-cycle, aggregated with security testing techniques applicable at every step [4].


Model-Based Security Testing
Code-Based Testing and Static Analysis
Penetration Testing and Dynamic Analysis
Security Regression Testing
Analysis
Design
Development
Deployment
Maintenance
Requirements
Design Models
Code
Running System

Model Based security testing

OWASP proposes Model Based security testing as security testing activity after the analysis phase and before the design phase. Model Based security testing consists in the following steps.

- This technique first constructs a model of the behavior of the software (and/or its environment) in order to later derive test cases from this model. The behavior of the system may be described in many manners including activity diagrams, sequence diagrams or simply input/output matrices [33].
- Then, security requirements (based on the security properties of the system's specification) are defined using a model-dedicated syntax.
- The next step is to generate automatically test cases derived from the model. That means execution traces of the model.

[7]OWASP: Open Web Application Security Project is an online community that aims at providing free resources in the field of web application security.

- Finally, the last step maps test cases from the model to real tests, that is tests launchable on the SUT.

Both positive and negative security testing may be performed at model level as one may model the behavior of the system but also the possible behavior of an attacker in order to derive test cases.

For instance, Model Based Security Testing has been experimented by Büchler *et al.* [35] for web app security testing. They take as assumption that a model of the web application and security properties are available. Then, they mutate this model using mutation operators introducing security-flaws. Next, they generate test cases by launching the model checker on the mutants. *Model checking* is a technique that consists in checking the satisfaction of a property in all the possible states of a model. Thus, the model checkers finds all sequences of operations that can uncover security problems in the mutants. Finally, those sequences of operations are semi-automatically mapped on real sequences, creating a selenium test for each sequence. We go through the details of their technique in chapter 3.

The advantage of model based security analysis is that it can be performed early in the development cycle therefore enabling early vulnerabilities discovery.

Code-based testing and static analysis

During the development phase in the life-cycle, OWASP [4] states the possible usage of code-based security testing and static analysis. Code-based security testing is mainly code analysis, *i.e* manually using code review or automatically using static analysis or another technique. In this section, we will concentrate on the static analysis.

Felderer *et al.* define static analysis as an attempt to automate code reviews. Hence, static security analysis automatically browses through the code seeking for fault patterns. As a vulnerability is a specific kind of fault, and static analyzer may thus be specialized for finding vulnerability patterns. Of course, there is an infinity of vulnerability patterns to be encoded in the static analyzer so one cannot expect this tool to be 100% accurate. However, experience has shown that this technique may effectively find many vulnerabilities [36]. Static analysis can be applied at source code or byte code level, the idea is that static analysis never runs the code. Though, the method also allows semantic checks, meaning an analysis of how the program should run, by the lecture of its code.

The advantages of static analysis for security testing is that it is relatively cheap and it can be performed quite early in the development cycle, as soon as a piece of code is available. Its drawbacks are that it requires human intervention, as a consequence of its static nature. Hence, because it does not run the code, uncovered vulnerabilities may not always lead to a failure. This problem is known as the *false positive problem.* Moreover, as we stated before, the fact that it does not find a vulnerability does not mean that the vulnerability is not there. This is known as the *false negative problem.*

We present a vulnerability static analyzer that was very helpful in our research in section 2.3.3.

Dynamic Analysis and Penetration testing

In the previous section, we presented statical ways of evaluating a system's security. In this section, we will focus on dynamic security testing, running the code in order to assess its security.

OWASP proposes to perform dynamic security testing after the deployment phase. This specification is likely due to the fact that its goal is to test security in its real environment. Moreover, we also hypothesize that security protections are implemented in order to be additive (minimizing the breaches in between each protection), so that there is no sense to test dynamically the security of a system before it is deployed.

Penetration Testing is a dynamic testing technique specialized in security testing. NIST [6] defines penetration testing as security testing the system in its environment, with the help of a penetration tester. A *penetration tester* is a practitioner performing real-world attacks on the system. Because the system is tested in its environment of deployment, one must always obtain the approval of the organization for which the system was built before performing penetration testing. Hence, the organization must differentiate the pen tester's activity from a real attack. NIST divides penetration testing in four phases.

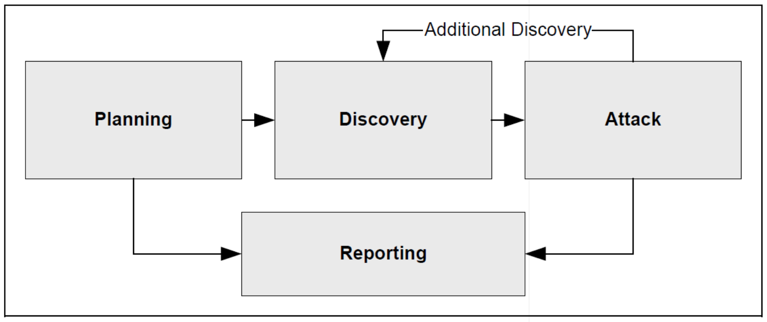
1. **Planning:** The first phase is to plan how the testing will be performed and acquire the organizations' approval. All the tests are defined and documented, explaining which part of the system is targeted and what is the expected behavior, for instance.
2. **Discovery:** The phase is divided in two parts. First, the pen tester discovers the *attack surface* of the system under test. That is gathering information about the system like its open ports, the services it runs, the versions of API's it utilizes, the OS it runs on, etc. The goal is to enumerate all the potential targets. Then, the pen tester compares this set to attacks' needs, in order to forecast what attack he will perform in the next phase.

3. **Attack:** During this phase, the pen tester simply performs the attacks he planned on the last phase. An attack consists in sending a set of malicious payloads to the target. If an attack succeeds, the found vulnerability is exploited in order to pursue the attack and access to more information or privileges on the target. The idea behind this is to understand the risks of the found vulnerability. The "expanded attack surface" is fed back to discovery phase in order to continue the process.

4. **Reporting:** This phase is performed in parallel with the three other phases. This phase consists in reporting the identified vulnerabilities, rank those in terms of severity, and give an advice on how to mitigate the vulnerabilities that are exploitable.

The next figure illustrates the process of penetration testing 2.7.

Figure 2.7: The process of penetration testing [6].



Vulnerability scanning can be seen as automatic pen testing. Hence, it consists in using tools called *Vulnerability scanners*, automatically sending malicious payloads to the SUT. Usually, vulnerability scanners are used by pen testers, benefiting of their experience in order to use the tools effectively.

Fuzzing consists in automatically sending "random" inputs to the SUT in order to make it crash or to launch an unintended behavior. The idea behind fuzzing is to test the behavior of the SUT under non intended inputs. We quote random because, as the technique was originally sending random inputs, modern techniques may also rely on the analysis of the SUT's code or on the smart choice of the inputs. Fuzzers may be compared in terms of effectiveness by comparing the number of un-intended behaviors they trigger on the SUT, in a given amount of time. In order to perform this comparison, one needs sets of documented vulnerable systems. We will come back to this last technique in the next chapters.

Security Regression Testing

The technique consists in applying traditional regression testing on the security requirements of the system. Hence, *regression testing* intends to create a high-coverage test suite, aiming at giving confidence that the SUT is legit to its specification. The test suite is then used during the maintenance phase by launching this suite after the introduction of a modification, in order to gain confidence that the modification did not alter the system's conformance to the original specification.

In security testing techniques performed during the software development life-cycle, OWASP advises to create security regression test suites to support maintenance. By this mean, it helps the software developers not to introduce vulnerabilities while modifying the system. The security regression test suite can take many forms, from the set of unit tests to the report of the pen testing phase. The idea is to have, again, a coverage measure and to automate as much as possible the testing activity.

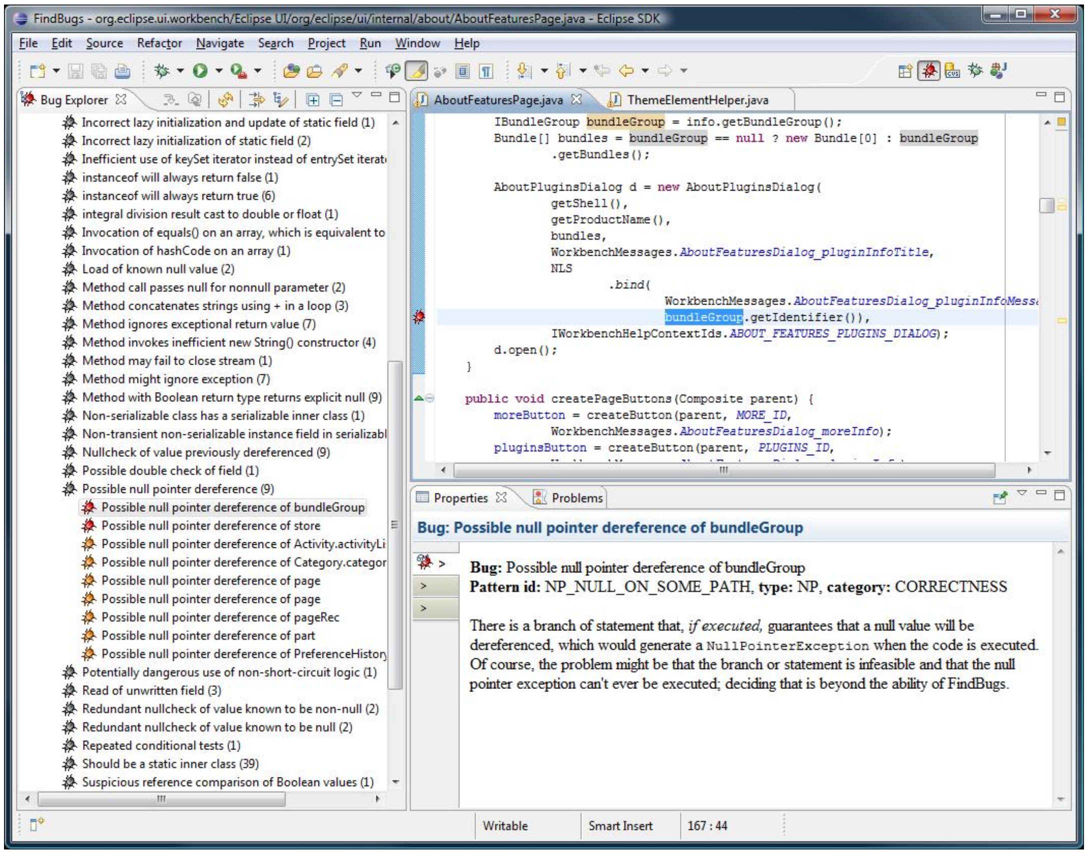
2.3.3 Used security testing tool: Findbugs-sec

In this section, we expose a real-world static analyzer aiming at finding vulnerabilities in JAVA code, that we utilized in our experimentation. This gives us the opportunity to illustrate the sub-section 2.3.2, that presented static analysis and code based security testing techniques.

FindBugs [37] is an open source static analyzer for JAVA, running at byte-code level. It is very popular as it has been downloaded more than a million times. The tool has been developed and is currently still maintained by the University of Maryland, USA. The tool reports almost 300 different fault patterns [38] and offers the possibility to create custom plug-ins. Many fault patterns are discovered using state machines on the class-files or by using the visitor pattern on method byte-codes. The static analyzer is, among others, used by Google. The organization runs it over any code being modified [38].

FindBugs is available in various forms, *e.g* in a maven plugin, in an eclipse plugin (figure 2.8), an IntelliJ plugin, an Android Studio plugin, a NetBeans plugin, in an Ant task, by using the command line or by using its GUI (figure 2.9). The tool is also highly configurable, enabling us to use filters or configure its sensitiveness to specific bug patterns. All its utilization possibilities are documented in a user manual.

Figure 2.8: Findbugs eclipse plugin illustration of usage.


FindBugs - org.eclipse.ui.workbench/Eclipse UI/org/eclipse/ui/internal/about/AboutFeaturesPage.java - Eclipse SDK

Bug Explorer

- Incorrect lazy initialization and update of static field (1)
- Incorrect lazy initialization of static field (2)
- Inefficient use of keySet iterator instead of entrySet iterat
- instanceof will always return false (1)
- instanceof will always return true (6)
- integral division result cast to double or float (1)
- Invocation of equals() on an array, which is equivalent to
- Invocation of hashCode on an array (1)
- Load of known null value (2)
- Method call passes null for nonnull parameter (2)
- Method concatenates strings using + in a loop (3)
- Method ignores exceptional return value (7)
- Method invokes inefficient new String() constructor (4)
- Method may fail to close stream (1)
- Method might ignore exception (7)
- Method with Boolean return type returns explicit null (9)
- Non-serializable class has a serializable inner class (1)
- Non-transient non-serializable instance field in serializabl
- Nullcheck of value previously dereferenced (9)
- Possible double check of field (1)
- Possible null pointer dereference (9)
  - Possible null pointer dereference of bundleGroup
  - Possible null pointer dereference of store
  - Possible null pointer dereference of Activity.activityLi
  - Possible null pointer dereference of Category.categor
  - Possible null pointer dereference of page
  - Possible null pointer dereference of page
  - Possible null pointer dereference of pageRec
  - Possible null pointer dereference of part
  - Possible null pointer dereference of PreferenceHistory
- Potentially dangerous use of non-short-circuit logic (1)
- Read of unwritten field (3)
- Redundant nullcheck of value known to be non-null (2)
- Redundant nullcheck of value known to be null (2)
- Repeated conditional tests (1)
- Should be a static inner class (39)
- Suspicious reference comparison of Boolean values (1)

AboutFeaturesPage.java | ThemeElementHelper.java

```
        IBundleGroup bundleGroup = info.getBundleGroup();
        Bundle[] bundles = bundleGroup == null ? new Bundle[0] : bundleGroup
                .getBundles();

        AboutPluginsDialog d = new AboutPluginsDialog(
                getShell(),
                getProductName(),
                bundles,
                WorkbenchMessages.AboutFeaturesDialog_pluginInfoTitle,
                NLS
                        .bind(
                                WorkbenchMessages.AboutFeaturesDialog_pluginInfoMess
                                bundleGroup.getIdentifier()),
                IWorkbenchHelpContextIds.ABOUT_FEATURES_PLUGINS_DIALOG);
        d.open();
    }

    public void createPageButtons(Composite parent) {
        moreButton = createButton(parent, MORE_ID,
                WorkbenchMessages.AboutFeaturesDialog_moreInfo);
        pluginsButton = createButton(parent, PLUGINS_ID,
```

Properties | Problems

Bug: Possible null pointer dereference of bundleGroup

Bug: Possible null pointer dereference of bundleGroup
Pattern id: NP_NULL_ON_SOME_PATH, **type:** NP, **category:** CORRECTNESS

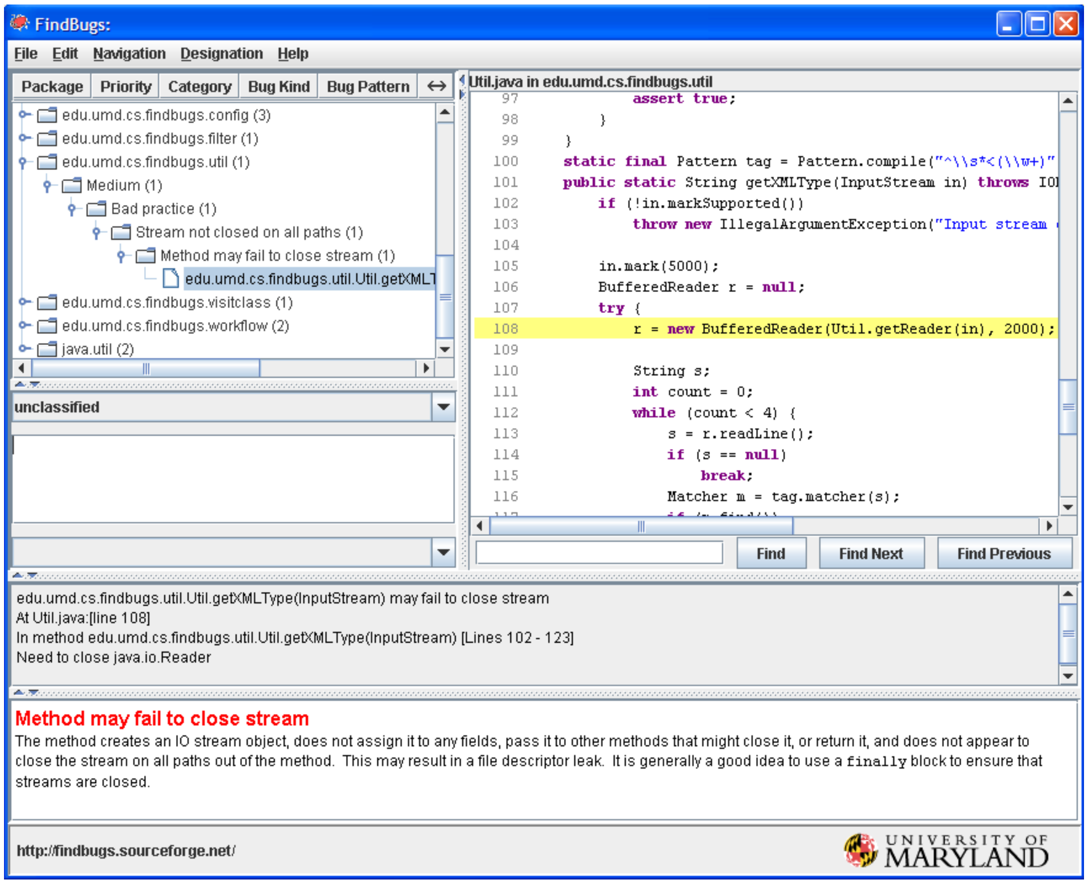
There is a branch of statement that, *if executed,* guarantees that a null value will be dereferenced, which would generate a `NullPointerException` when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception can't ever be executed; deciding that is beyond the ability of FindBugs.

Writable | Smart Insert | 167 : 44

During our research, we used FindBugs but more specifically one of its most popular plugin, Findbugs-sec [19]. The plugin enables Findbugs to highlight more than 110 different vulnerabilities. Therefore, Findbugs can, by activating this plugin, become a security static analyzer (see sub-section 2.3.2).

The plugin is open source, and was originally developped by Philippe Arteau [19]. All the patterns the plugin highlights are described in its documentation. A great advantage of the tool is that every vulnerability it uncovers is justified by a CVE or NIST recommendation. This last statement enables us to use the tool's documented vulnerabilities in order to create real-world mutation operators. We detail this process in chapter 4.

Figure 2.9: Findbugs' GUI illustration of usage.


FindBugs:
File Edit Navigation Designation Help
Package Priority Category Bug Kind Bug Pattern
edu.umd.cs.findbugs.config (3)
edu.umd.cs.findbugs.filter (1)
edu.umd.cs.findbugs.util (1)
Medium (1)
Bad practice (1)
Stream not closed on all paths (1)
Method may fail to close stream (1)
edu.umd.cs.findbugs.util.Util.getXMLT
edu.umd.cs.findbugs.visitclass (1)
edu.umd.cs.findbugs.workflow (2)
java.util (2)
unclassified
Util.java in edu.umd.cs.findbugs.util
Find
Find Next
Find Previous
http://findbugs.sourceforge.net/
UNIVERSITY OF MARYLAND

```
 97             assert true;
 98         }
 99     }
100     static final Pattern tag = Pattern.compile("^\\s*<(\\w+)"
101     public static String getXMLType(InputStream in) throws IO
102         if (!in.markSupported())
103             throw new IllegalArgumentException("Input stream
104 
105         in.mark(5000);
106         BufferedReader r = null;
107         try {
108             r = new BufferedReader(Util.getReader(in), 2000);
109 
110             String s;
111             int count = 0;
112             while (count < 4) {
113                 s = r.readLine();
114                 if (s == null)
115                     break;
116                 Matcher m = tag.matcher(s);
```

edu.umd.cs.findbugs.util.Util.getXMLType(InputStream) may fail to close stream
At Util.java:[line 108]
In method edu.umd.cs.findbugs.util.Util.getXMLType(InputStream) [Lines 102 - 123]
Need to close java.io.Reader

Method may fail to close stream

The method creates an IO stream object, does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a `finally` block to ensure that streams are closed.

Chapter 3

Mutation Security Testing

This chapter develops the state of the art of the research problem we briefly described in the introduction. Therefore, we begin by enunciating in details the research problem we try to handle. Then, we supply the related work partially handling the problem or related to it.

3.1 Problem statement

In chapter 2, we motivated the usefulness of mutation testing in the tests' creation process and stated several security testing techniques. Yet, the relation between security testing and mutation analysis remains to be explored. Indeed, existing mutation operators, especially those used by the JAVA mutation testing tools [23], are restricted to simple syntactic alterations and faults. Because these syntactic modifications are unlikely to (effectively) introduce security defects, we believe that mutation security testing requires the creation of specialized operators. Thus, we formulate the following hypotheses that guided our research.

Non security-aware mutation operators are unlikely to incidentally generate vulnerable mutants.

Although designing mutation operators that introduce vulnerabilities has been explored before (see next section 3.2), it seems that there is no set of security-aware mutation operators at code level.

Mutation operators specialized in introducing vulnerabilities at code level were never experimented before.

In this thesis, we solve the issue by designing these security-aware mutation operators, introducing vulnerabilities rather than simple faults. For demonstration, we design these operators for the JAVA language as this language is very popular to implement security-aware systems.

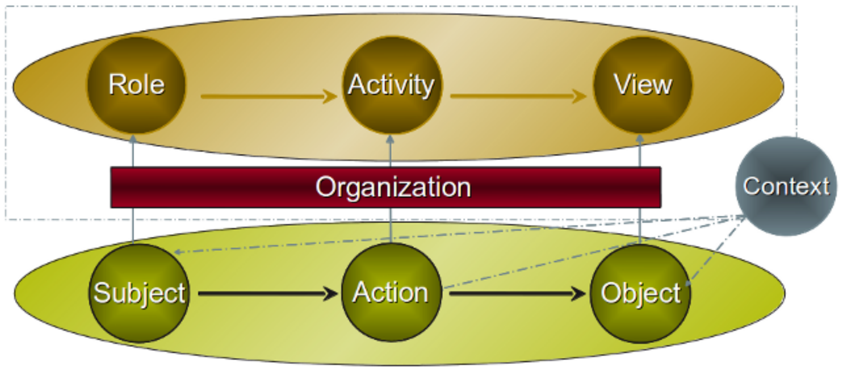
In the next section, we present the related work, the different ways of designing security mutation operators that we could find and demonstrate, for each of them, their non applicability in our context.

3.2 Related work

3.2.1 Mouehli *et al.*: Mutating access control models

In this work, the authors [39] aim at giving a criterion to test the access control policy of 3-tier applications written in the *ORBAC language (ORganisation Based Access Control)*. *ORBAC* is a language specifically designed to write access control policies. It allows programmers to define *Permissions*, *Prohibitions*, or *Obligations*. All these rules are defined as 5-uples of entities: the organization, the role, the activity, the view and the context. Usually, rules specifying access control policies are defined using 3-uples of *Subjects*, *Actions* and *Objects*. The 5-uples rules used in ORBAC are actually these exact 3-uples to which the language concatenates an organization and a context in which the rule is valid (see figure 3.1).

Figure 3.1: Illustrating how the concepts of Subjects, Actions and Objects are respectively related to Roles, Activities and Views in order to define security rules in ORBAC [7].



The authors provide examples of those rules for a library management system. In this system, an access control requirement can be *Users are allowed to borrow a book only when the library is opened*. The rule is defined as : `Permission(Library, Borrower, BorrowerActivity, Book, WorkingDays)`. The rule contains a specific library as organization entity, the role of a borrower, the activities that a borrower may perform in the system, the view of a book and the context of the working days.

Rules specifying access control policies may sometimes be conflicting, and ORBAC has the advantage to propose a processing tool (MotOrBAC [40]) aiming at discovering those conflicts.

The authors state that there is no automatic way to generate secure code enhancing a security policy. Therefore, programmers should implement themselves the *Permissions*, *Prohibitions* and *Obligations* in the code of the application. They also state that because the implementation of the access control rules is manual, it must imperatively be tested. In order to perform this, the authors propose mutation operators, mutating the rules that compose the access control policy. Using the generated mutants, the testers discover which abstract cases they should cover. Mouelhi *et al.* state that the advantage of the method is that the faults introduced specifically aim at testing the security of the SUT. However, the drawback of the method is that imagining the mapping between the mutations and errors in the implementation, a relatively hard task, is left to the tester. They mitigate this problem by proposing a mapping tool between the ORBAC access control rules and the code implementing each rule.

The mutation operators are diverse: they can change the organization, role, activity, view or context in a rule. They may also invert Permissions, Obligations or Prohibitions. Mouelhi *et al.* also implemented a mutation-testing tool, as an additional part of *MotORBAC*.

The method is interesting but its objective is really different from what we would like to achieve. Hence, we would like to generate vulnerabilities lying at code level in order to be able to focus on concrete vulnerabilities when implementing a security test-suite. The current method is limited to generating those vulnerabilities at model level and lets the tester imagine what those abstract flaws would look like at code level. Also, model-based mutation introduces different faults than those introduced a the code-level [41].

3.2.2 Dadeau *et al.*: Mutating high-level security protocols

In this research, the authors investigate the possibility of verifying security properties (like secrecy, authentication or data integrity) in the implementation of security protocols. Those protocols are defined in the high-level security protocol language *HLPSL* and are specifically aiming at establishing a trust-full communication link between actors[1].

[1]Actors are participants in a protocol. In HLPSL, an actor is represented by a state-transition system where transitions are triggered by received messages and where actions trigger messages being sent.

The approach proposes to mutate the model of the protocol-under-test, in order to generate abstract test cases, by validating automatically the generated mutants. The abstract test cases obtained should then be followed as a criterion for implementing test cases on the implementation of the protocol. Thus, the authors introduce new mutation operators that aim at introducing leaks in the models of the protocol and at representing real world implementation faults.

The mutation operators introduce the following faults: replacing in a protocol specification every encryption by a xor-encryption, decomposing information blocks in encrypted messages, injecting the use of the same public key by two different actors, suppressing the verification of message provenance in the behavior of an actor, omitting hash functions and permuting the order of information blocks in a message. All the operators are justified by providing an example of protocol where their usage leads to a security failure.

Aside from the fact that the method is applied at model level and that we are focusing on code level, it also is not applicable in the context of our research. Hence, the authors propose security-aware mutation operators for protocols whereas aim at introducing security faults in webapps.

3.2.3 Büchler *et al.*: Mutating abstract model of web applications

In this work, the authors propose a semi-automatic testing technique for web applications [8]. Their methodology assumes that the penetration tester possesses a model of the webapp under test and that the security properties (*e.g* confidentiality, authenticity, etc.) of the model under test are available. Thus, the model satisfies its security properties.

The methodology is composed of four essential steps (see figure 3.2).

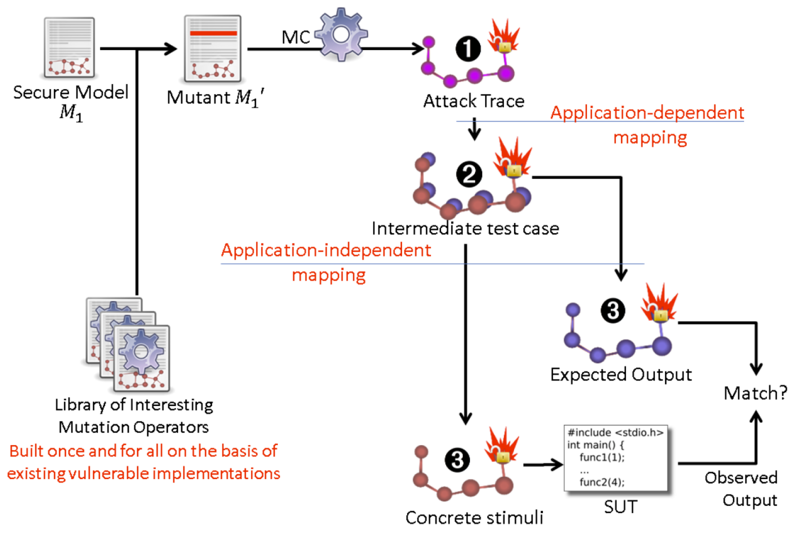
1. First, mutate the model using specific mutation operators specifically designed to introduce vulnerabilities in the model. The mutation operators are associated to several security properties that they threatened before, *i.e* in another insecure model, enabling the methodology to use a mutation operator only if it is likely to introduce a vulnerability in the SUT's model. The authors list themselves the security properties that each mutation operator threatens and gather those from the analysis of several vulnerable applications. This step generates several vulnerable mutants, *i.e* insecure models that represent faulty implementations of the SUT, where a fault led to the injected vulnerability.
2. Second, run a model checker on the vulnerable mutants. Each mutant is run with the

security properties of its original model. The model checker outputs a trace for every mutant where a security property is violated (item 1 on the figure). Every trace is taken as an abstract test case that should be mapped on a concrete test case. This step thus generates a set of abstract test cases.

3. Next, translate each abstract test case in a concrete test case by using a 2-step mapping (item 2 and 3 on the figure).

4. Finally, run the concrete test cases on the real system by using an automatic test execution engine. This engine may ask help from a penetration tester when it cannot perform an action. These actions are the abstract steps of an abstract test case that could not be mapped to concrete actions.

Figure 3.2: Büchler *et al.* methodology. "Overview of the testing process" [8]


Secure Model M_1
Mutant M_1'
MC
1
Attack Trace
Application-dependent mapping
2
Intermediate test case
Application-independent mapping
3
Expected Output
Match?
Library of Interesting Mutation Operators
Built once and for all on the basis of existing vulnerable implementations
3
Concrete stimuli
SUT
Observed Output

```
#include <stdio.h>
int main() {
    func1(1);
    ...
    func2(4);
```

The model of the application is built using the abstract messages that it supports from the user like the log in to the application, view the other users' profile, etc. Therefore, an attack trace generated by the model checker will consist in a sequence of abstract messages sent by the user to the server. Both the model and the properties are defined in AVANTSSAR Specification Language (ASLan++), an input language for model checkers that is dedicated to security analysis. The 2-step mapping first consists in the translation of the message-sequence attacks into an intermediate language (called *WAAL*: Web Application Abstract Language[2]) that is independent from the webapp under test. Then, the WAAL attack trace is mapped to a concrete sequence of actions that are described using the *Selenium framework*[3]. The help of the penetration tester will be needed for actions that are not supported by the framework, that fact leading to the semi-automatic nature of the methodology.

The authors explain that their mutation operators are related to real vulnerabilities that they found in unsecured webapps. Hence they were built by the following steps:

1. First, analyze a set of vulnerable web applications and gather their vulnerabilities.

2. Next, build or obtain a secured model of these web applications, *i.e* the model of the web application as it should have been implemented. Build and obtain the security properties of the models.

3. Then, link each real world vulnerability to the security properties that it violates.

4. Finally, build a mutation operator working at model level that introduces, at model level, an equivalent to each real world vulnerability considered. Then, verify that every mutation operator is actually capable of generating a vulnerable mutant. That step is realized by generating a mutant in order to run it on the model checker and verify that the security properties it should violate are indeed violated. The whole process furnishes to the authors a set of t-uples *(mutation operator, security property violated by the operator)*.

The authors evaluate their operators on a case study. Hence, they state that the appliance of their operators helped them at finding vulnerabilities in an unsecured web application, *WebGoat*[4].

[2]WAAL is an abstract language that enables the specification of actions that a user must perform using the browser and the specification of verifications to be performed on the server's response. WAAL is used here to describe a set of actions to perform by using the browser. Its interest as a layer between the trace given by the model checker and the concrete test is to define the exact actions and verifications to perform in order to conduct the test. This enables the penetration tester to perform the test himself by using the WAAL specification of the test.

[3]Selenium is an open source portable testing framework for web applications. It provides support in several programming languages including C#, JAVA, PHP, Python and Scala. The tests consist in actions performed at browser level and in the analysis of the browser's responses.

[4]WebGoat is an intentionally vulnerable web application developed by OWASP. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

Again, the method relies on operators working at model level. However, the authors explain that they possess a mapping between real world vulnerabilities and vulnerabilities at model level. This mapping could be interesting to analyze, in order to inject those real world vulnerabilities. Unfortunately, Büchler *et al.* do not detail their operators.

3.2.4 Mutating C programs by introducing memory manipulation issues

Memory faults may lead to a large diversity of vulnerabilities. Nanavati *et al.* state that memory faults have led before to vulnerabilities such as uninitialized memory access, buffer overruns, invalid pointer access, beyond stack access, free memory access or memory leaks in C applications. Moreover the authors add that these memory faults have been ranked in the top 25 most dangerous programming errors by CWE SANS. Therefore, the authors introduce Memory Mutation Operators (*MeMOs*) capable of injecting common memory faults in C programs in order to enable mutation testing for security. Thus, Nanavati *et al.* focus on one specific class of vulnerabilities when testing the security of an application. Moreover, the authors prove the applicability of their mutation operators by killing the mutants they generate on 16 subject C programs. Their operators are composed of three categories: uninitialized memory access, faulty memory allocation and faulty heap management.

This approach towards security-aware mutation operators in C has also been experienced by Shahriar and Zulkernine [42]. Hence, Shariar and Zulhernine propose 12 mutation operators that mutate ANSI C standard library functions, modify buffer size arguments in ANSI C standard library functions, mutate format strings, increase buffer variable sizes and remove null character assignment statements. These operators are designed for their capacity in introducing vulnerabilities that are potentially exploitable to perform a *Buffer Overflow*[5] attack. The authors demonstrate the applicability of their mutation operators on four case studies, by killing most of the mutants they generate using their operators.

Ghosh *et al.* [43] also mutate C programs in order to identify software chunks that may threaten security properties when mutated (altered). The idea is to leverage those code snippets to the programmer in order to raise its awareness about those locations where a security property violation is possible. Moreover, their goal is to show the programmer or analyst how badly the software could behave at specific locations, if it was altered. The authors justify their approach by stating that rather than searching for flaws in software, they simulate the effects

[5]A Buffer Overflow is an overflow in data buffers that might lead to the corruption of neighboring variables [42].

of flaws in software and leverage locations where fault-tolerant mechanisms might be needed.

Their methodology has been implemented in a tool named the Fault Injection Security Tool (FIST). It performs white-box dynamic security analysis of the SUT using program inputs, fault injection and security assertion verification for programs written in C and C++. Hence, it consists in running the mutants with legal, random, and illegal inputs and controlling their state for violations of security assertions that are *a priori* provided by the analyst. If a security assertion is violated, the fault and the input that triggered the violation are recorded for further analysis.

In their study, Ghosh *et al.* propose several types of mutation operators. They name their types of operators *buffer overflow, data corruption, string manipulation and fault composition.* The buffer overflow operator overwrite the return address of the stack frame in which the buffer is allocated with the address of the buffer itself. Also, the buffer is filled with machine instructions by the operator. These machine instructions will thus be run whenever the program reaches the return point that was modified, resulting in the corruption of the program state and the possible violation of a security assertion. The data corruption operators consist in simple syntactic changes in booleans, characters, strings, integers and doubles. For instance, these operators apply a negation to a boolean value or substitute a character with a random other, randomly chosen in the ASCII table. The string manipulation operators also consist in some rather simple syntactic changes. For instance, these operators truncate strings or substitute a string with a randomly generated other. The last type of operator consists in the possibility to compose faults, by composing the previously introduced mutation operators.

The data corruption operators are very close to standard mutation operators, and we thus will not need to re-implement those in our study. Moreover, in all the operators they introduce, only the buffer overflow operator seems to specifically aim at introducing vulnerabilities.

In every explored method, the authors have found an effective set of mutation operators that is actually capable of injecting vulnerabilities at code level. However, the vulnerabilities they inject are not applicable in our context. Hence, all of their mutation operators consist in the introduction of memory manipulation faults in C programs. As all these memory manipulations make use of memory functions that are specific to C, these operators are not applicable in our context of JAVA Web Applications.

In the next section, we summarize the state-of-the-art that we just explored in the sections 3.2.1 to 3.2.4.

3.2.5 Conclusion

Using mutation for security purposes was explored at the model-level by Mouehli *et al.* [39] where the authors mutate access control models to qualify security test suites. Operators change user roles and allow actions, deleting policy rules or modify their application context. Dadeau *et al.* defined operators that introduce leaks in a high-level security protocol [44]. Büchler *et al.* considered mutating the abstract model of a web application by removing authorization checks and un-sanitizing data [8], but they do not detail the operators. Though such operators also take inspiration from actual vulnerabilities, model-based and code-based mutation exercise different faults [41], these works are thus not directly applicable to ours.

To the best of our knowledge, there is no set of security-aware mutation operators at code level that suits our context. Perhaps the closest related work is that of Nanavati *et al.* [45], Shahriar and Zulkernine [42] and Ghosh *et al.* [43] that defined mutation operators related to the memory related faults. As explained in section 3.2.4, all these operators introduce memory manipulation issues in C programs (such buffer overflows, uninitialized memory allocations, etc.), which may be exploited by security attacks. As these operators make heavy use of memory allocation primitives, specific to the C language, they are not applicable to our context (JAVA webapps) and are not targeted at generating a wide range of security issues.

Chapter 4

Security aware mutation operators

In this chapter, we address our research problem by stating possible security-oriented mutation operators. First, we present the procedure we followed to shape the operators in section 4.1. Then, we articulate the abstract definition of these operators in section 4.2. Finally, we also give technical details about their implementation in section 4.3.

4.1 Mutation operators definition procedure

This section presents the process we followed to design our mutation operators.

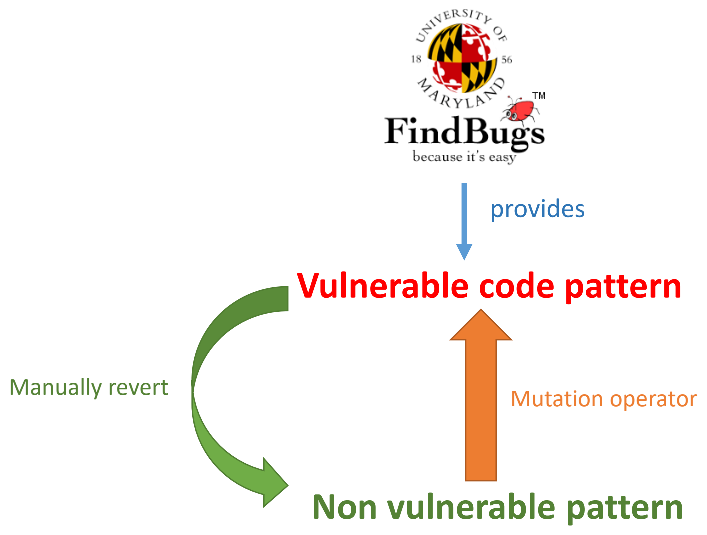
Security related issues have received little attention by the mutation testing literature. As a result, it lacks operators that introduce security bugs. While security bugs are more or less well studied, there is no clear definition that we could use. Therefore, for the purposes of this study we will use the following definition: a *security bug* is a piece of code that can lead to one or several vulnerabilities in an application.

Research on software security developed a number of techniques to identify vulnerabilities in source code. We have seen such a (effective) technique, static analysis in chapter 3. As we explained, static analyzers aim at identifying occurrences of problematic code patterns. Such tools applied to security detect security bugs by highlighting potentially vulnerable code patterns based on common vulnerability patterns. In view of this, we propose to leverage their knowledge and gather a set of common security bugs, which we can turn into injectable faults. These faults can form our mutants and support security testing.

By gathering the security patterns supported by known static analysis tools, we can identify certain types of security related faults. Unfortunately, these patterns only detect the presence of a potentially vulnerable code and not the needed transformation to inject a vulnerability. Indeed, a security mutation operator identifies a non-vulnerable code pattern and turns it into a vulnerable one.

Therefore, in order to define our operators, we transformed every occurrence of a vulnerable pattern we could gather in its non-vulnerable functional equivalent version. This was not a trivial task as it required manual analysis and comprehension of the vulnerability classes.

Figure 4.1: Process followed to design security-aware mutation operators.



For the purposes of the present study, we used the security patterns of a well-known static security bug analyzer, named *FindBugs-sec plugin*. All the patterns we used are described in the plugin's documentation [19]. We believe that these patterns are suitable for our purposes as most of them form real-world security bugs, this being shown with CVE[1] and NIST[2] references. We detail our operators in the next section. We illustrate our process with figure 4.1.

The next step was to implement those operators in a JAVA mutation tool. After investigating which tool to use (see chapter 2), we implemented our operators in a popular open source mutation tool called *PITest* [3]. We explain briefly the implementation of our operators in PIT and, more generally, how mutation operators are implemented and used in the tool in section 4.3.

While implementing our operators, we manually reviewed them. In order to check the viability and the correct implementation of our operators, we generated by their use several vulnerable sample programs. By viability checking, we mean verifying that the vulnerabilities did not compromise the class in which they were introduced.

4.2 High level definition of the operators

Table 4.1 couples acronyms to a short description for each of the security-aware mutation operators we propose.

In this section, we will give for every operator its application *Context*, the vulnerability it is trying to generate in the *Goal* section, and a short specification of its implementation in the *Specification* section. Table 4.1 also sorts the different operators by their nature and by their implementation design. We present the operators in the next sections.

Replacing secured objects with unsecured:

In this section, we present the mutation operators that replace secured objects with unsecured.

Use predictable pseudo random number generator (UPPNRG).

Context: In secure-aware contexts and more specifically in encryption, developers need Pseudo Random Number Generators for features like salts and keys generation. In order to avoid prediction easing an undesired decryption, the PNRGs used in this context should be unpredictable.

[1]CVE: Common Vulnerabilities and Exposures : is a database of known information-security vulnerabilities and exposures maintained by the MITRE corporation.

[2]NIST: The National Institute of Standards and Technology publish technical reports about many subjects in IT. Moreover, they often publish reports regarding common vulnerabilities and good practices for security engineers.

Table 4.1: Security-aware Mutation Operators

Acronym	*Name*
Replacing secured objects with unsecured:	
UPPRNG	USE_PREDICTABLE_PSEUDO_RAND_NUM_GEN
UWMD	USE_WEAK_MESSAGE_DIGEST
REIS	REMOVE_ENCRYPTION_IN_SOCKET
PSQLI	PERMIT_SQL_INJECTION
Removing security features on java objects:	
XMLPVXEE	XML_PARSER_VULNERABLE_TO_XEE
XMLPVXXE	XML_PARSER_VULNERABLE_TO_XXE
UC	UNSECURE_COOKIE
RHTTPOFC	REMOVE_HTTPONLY_FROM_COOKIE
Weakening encryption algorithms:	
URSAWSK	USE_RSA_WITH_SHORT_KEY
UBFWSK	USE_BLOWFISH_WITH_SHORT_KEY
Removing standard sanitization functions:	
RPTS	REMOVE_PATH_TRAVERSAL_SANITIZATION
RRS	REMOVE_REGEX_SANITIZATION
Replacing strong with weak encryption algorithms:	
UDESISE	USE_DES_IN_SYMMETRIC_ENCRYPTION
UECBISE	USE_ECB_IN_SYMMETRIC_ENCRYPTION
Removing authentication mechanisms:	
RHNV	REMOVE_HOST_NAME_VERIFICATION

Goal: The `UPPNRG` operator's goal is to make the application vulnerable to predictable random number generator exploiting attacks. The success of these attacks can lead to various security leaks regarding the confidentiality, authorization, *etc.* of the application.

Specification: The operator trades unpredictable PRNGs' usages for predictable PRNGs' usages.

Use weak message digest (UWMD).

Context: Message digests, or hashing functions, are very often used to assure the integrity of received data. They may also be used in authentication mechanisms. However, some hash functions are weak because of their high collision degree: in this case, for a hashed string, a malicious user can easily craft another string producing the same hash.

Goal: The `UWMD` operator introduces a vulnerability in integrity checking of received data or in authentication mechanisms by replacing a secured hash function (*i.e*, SHA-256) use with a unsecured hash function (MD5).

Specification: It recognizes hash function utilization and replaces those by MD5 usage.

Remove encryption in socket (REIS).

Context: Web applications very often need to communicate with users using a encrypted channel. Indeed, confidential data such as passwords or e-mail addresses should never be exchanged unencrypted. A safe channel is commonly implemented using SSL on HTTP.

Goal: The `REIS` operator weakens the confidentiality of exchanged data, exposing the application to a confidentiality leak.

Specification: It removes the use of SSL in https sockets.

Permit SQL injection (PSQLI).

Context: SQL injections exploit the fact that the web application uses input(s) from the user to build an SQL query that will be executed by a Data Base Management System (DBMS). The idea of the SQL injection is to inject SQL code in the inputs that are used to build the query, in order to maliciously alter the database and/or get access to private information. To prevent these attacks, JAVA APIs provide methods to prepare/encode queries and send them without their external inputs to the DBMS, requiring these inputs separately from the user. The DBMS that received the encoded query waits for the arrival of the inputs in order to finalize the construction of the query and execute it. Using this solution, it is not possible anymore for an attacker to play with the SQL-syntax to create tainted queries as the semantic of the query is computed without taking account of the inputs values.

Goal: The `PSQLI` operator tries to weaken the web application's protection against SQL-injection attacks to expose it to various security leaks potentially threatening its confidentiality, integrity, and authentication mechanisms.

Specification: It detects usages of SQL injection-proof APIs for query executions and replaces such usages by unsecured APIs query executions.

Removing security features on JAVA objects:

In this section, we detail the mutation operators that remove security features on JAVA objects.

Make XML parser vulnerable to XML external entity expansion attack (XMLPVXEE).

Context: Web services often parse XML documents, to communicate with other web services in a standardized way. A known attack is the *billion laughs* attack which is an instance of a denial of service attack on XML parsers that require them to exponentially expand the tree with dummy text ("LOL"). However, one can prevent this attack by enabling a standard security option on the XML parser.

Goal: The `XMLPVXXE` operator introduces a vulnerability in external XML parsers to expose the application to DOS attacks.

Specification: The operator disables standard security options of the XML parsers just before the XML parser begins parsing. It performs this task by identifying methods used on standard XML Java parsers, like `XMLReader` or `SAXParser` instances.

Make XML parser vulnerable to XML external entity attack (XMLPVXXE).

Context: For this operator, we add to `XMLPVXEE`'s context that the attacker has access to the XML document parsing result. The hypothesis we added enables XXE attacks to be performed on the XML parser. Indeed, the attack's goal is to access to confidential (or unauthorized) files. Thankfully, XML parsers provide standard options restricting usage of commonly used XML tags in XXE attacks.

Goal: The `XMLPVXXE` operator aims at introducing a vulnerability to XXE attacks on XML parsers that are used by the application under test.

Specification: In a similar way as the `XMLPVXEE` operator, the `XMLPVXXE` operator disables XXE security options on XML parser objects. It disables this options just before a parsing begins, in the code of the application.

Unsecure cookie (UC).

Context: Cookies are defined by the HTTP protocol as pieces of information sent by the server to the client's browser. Some cookies can store secret values that prove a client's authentication and must therefore be encrypted using SSL during communication. Cookies are meant to be sent by the browser with each request from the client, disregarding the secured-nature of the communication. To make sure that a browser will not send a sensitive cookie in an unsecured HTTP communication by mistake, a *secure flag* can be set on the cookie, asking the browser to send this cookie only during HTTPS communications.

Goal: The `UC` operator allows to send sensitive cookies during unsecured HTTP communication. The mutation can lead to authentication leaks. Confidentiality and authorization leaks can also potentially be introduced by the operator.

Specification: It removes the call to standard methods setting the secure flag on cookies.

Remove HTTP-only flag from cookie (RHTTPOFC).

Context: Even if a cookie was sent using an HTTPS communication, web pages' scripts can access it on the client-side by asking the browser concrete access to the sessions' cookies. An attacker may get access to those cookies on the client-side by using a cross-site scripting (XSS) attack. To prevent this, cookies have an *HttpOnly flag* asking the client's browser to not share this cookie with scripts. Of course, the flag mitigates the risk and does not remove it, since it relies on the trust in the browser.

Goal: The `RHTTPOFC` operator exposes the web pages to such cookies confidentiality leaks.

Specification: It removes the call to the methods setting the http-only flag on cookies.

Weakening encryption algorithms:

In this section, we present the mutation operators that are weakening encryption algorithms.

Use RSA with short key (URSAWSK).

Context: RSA is an asymmetric encryption algorithm used in web applications to exchange confidential data. Over time, with the improvement of computation power, the RSA algorithm needs longer keys to keep the exchange secured and to resist to brute force attacks. NIST[3] recommends the RSA keys to be at least 2048 bits long.

Goal: The `URSAWSK` operator tries to weaken RSA encryption so that brute force attacks are possible, allowing confidential data to leak.

Specification: It detects the use of RSA encryption with a sufficient key size and sets its to 512 bits.

[3]http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf

Use Blowfish with short key (UBFWSK).

Context: Blowfish is a variable key size symmetric encryption algorithm five times faster than triple DES. Just like RSA, Blowfish could also be used with a insufficient key size (less than 128 bits).

Goal: The `UBFWSK` operator tries to weaken Blowfish encryption to expose the application to brute force attacks, thus allowing data to confidentiality leaks.

Specification: When the operator detects the usage of Blowfish with a sufficient key size, it sets the key size to 64 bits.

<u>Removing standard sanitization functions:</u>

In this section, we present the mutation operator that remove standard ways of sanitizing inputs coming from an external user.

Remove path traversal sanitization (RPTS).

Context: Web applications may often provide internal file access to external users. This feature is commonly designed by asking the user the name of the file he wants access to.

Goal: The `RPTS` operator introduces a vulnerability which allows a malicious user to enter a path where a file name was required. This attack may give him access to directories or files regardless of the webapp's file access policy.

Specification: The operator simply removes (input) file names sanitization functions, generally used to avoid this vulnerability.

Remove regex sanitization (RRS).

Context: Modern websites are following the idea of WEB 2.0, enabling the participation of external users to the content of a web page. Thus, web applications can store a lot of content coming from external users. To prevent storing malicious users' content, web applications commonly validate the inputs coming from external sources using regular expressions.

Goal: The `RRS` operator tries to introduce vulnerabilities in external input filters of a web application.

Specification: It detects regular expressions usages and replace them by a dummy expression, which is always true.

Replacing strong with weak encryption algorithms:

In this section, we explain the mutation operators that replace strong encryption algorithms usages with weak encryption algorithms usages.

Use DES in symmetric encryption (UDESISE).

Context: In secured web applications, symmetric encryption is often very valuable to exchange sensitive data with external users. Data Encryption Standard (DES) was a popular symmetric encryption algorithm that became nowadays sensitive to brute force attacks due to the great advances in computer performances. Therefore, web applications should prefer other symmetric encryption algorithms, like AES.

Goal: The idea of the `UDESISE` operator is to weaken the confidentiality of symmetrically encrypted data, exposing it to leaks.

Specification: It detects the usage of a symmetric encryption algorithm and replaces it with DES encryption. This operator requires to modify several JAVA code lines. Though, PIT's architecture wasn't designed for this kind of modifications. Therefore, `UDESISE` is still under review but our initial implementation provides promising results.

Use ECB in symmetric encryption (UECBISE).

Context: Symmetric encryption may be done using different modes, describing how the algorithm should encrypt a message, that is split in blocks of fixed size. The Electronic CodeBook (*ECB*) mode encrypts two identical blocks into two identical ciphered blocks, introducing redundancy in the encrypted message, which makes it easier for an attacker to decrypt the message.

Goal: The `UECBISE` operator tries to weaken the confidentiality of symmetrically encrypted data by easing its decryption using ECB mode.

Specification: It detects the usage of a symmetric encryption algorithm and replaces its mode by *ECB*.

Removing authentication mechanisms:

In this section, we present an operator removing authentication mechanisms that are present in the code of a Web Service for instance.

Remove host name verification (RHNV).

Context: A web application needing to authenticate its clients may verify their host names, usually after a successful SSL handshake.

Goal: The `RHNV` operator removes this authentication, making the application vulnerable to man-in-the-middle attacks.

Specification: It removes standard methods used to authenticate clients using their host names.

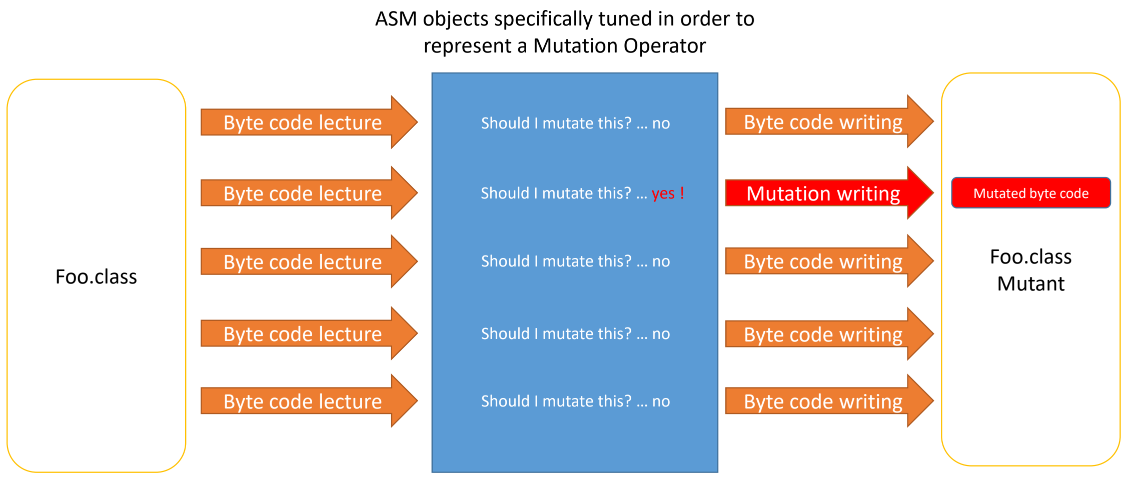
4.3 Implementation details

In this section, we give implementation details about PIT in order to support future research using the tool. Hence, PIT's mutation operators' design was not imagined to be able to perform complex mutations and we hope that this contribution will help those that are reluctant to use the tool at first glance. We also give more details about the precise behavior of our mutation operators, in order to support future improvements of our work.

PIT is a JAVA mutation testing tool working at byte-code level, *i.e* it does not have to compile its mutants but rather mutates the code of the program under test at byte-code level. The tool relies strongly on a byte-code manipulation library named ASM for its mutation operators' implementation. Hence, the library proposes JAVA objects that read a class byte-code and write what they have read in a buffer. However, these objects may be tuned in order to modify what they write on specific byte-code lecture. This enables to create mutation operators, that trigger mutations simultaneously while the reading a class and writing a mutant (see figure 4.2).

In PIT, the library was used to go through the byte-code of a class under test, identify method declaration parts of the class, and read line by line the code of the identified methods while triggering mutations on the go.

Figure 4.2: ASM objects that are reading `Foo.class`'s byte-code and triggering a mutation while the lecture is ongoing.



ASM[4] implements the features we just described in the following way.

First, it proposes to use a `ClassReader`, a class to which a programmer can pass byte-code by using the constructors of the class, *e.g* `ClassReader(byte[] b)` or `ClassReader(InputStream is)`. In PIT, the `ClassReader` is used to read the byte-code of the program under test.

The `ClassReader` extends the `ClassVisitor` class which follows the visitor design pattern, and that is used to visit byte-code JAVA classes. Moreover, the `ClassVisitor` owns some methods like `visit(int version, int access, String name, String signature, String superName, String[] interfaces)` to visit the header of a class, `visitAnnotation(String desc, boolean visible)` to visit the annotation of a class, or `visitMethod(int access, String name, String desc, String signature, String[] exceptions)` to visit a method of a class. The visit methods can be very fine grained. Hence, there is a visit method for almost any byte-code instruction composing a JAVA class. The attributes passed to a `visit` method are filled during the lecture of the byte-code of a class.

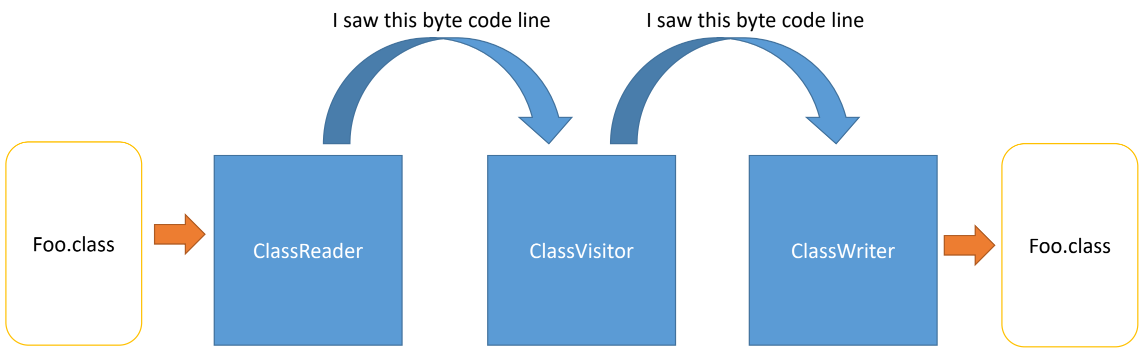
[4]The ASM JAVA byte code manipulation framework is detailed at http://asm.ow2.org/, the website provides several tutorials and a user guide. In this work, we used the fourth version of the library.

Next, it offers the possibility to connect a `ClassVisitor` to another one by providing the first with the reference of the other. In this case, whenever the first `ClassVisitor` visits some code by calling one of its `visit` method, it also calls the same `visit` method of the other. The real benefit of connecting `ClassVisitor`'s is to create a chain of visitors by the following way (see figure 4.3).

1. First, create a `ClassReader`. Then, connect one `ClassVisitor` to this `ClassReader`. Hence, while the `ClassReader` reads the class under test, the `visitX` methods of its parent are called. Therefore, the `ClassReader` sends anything it visits to the `ClassVisitor` to which it is connected.

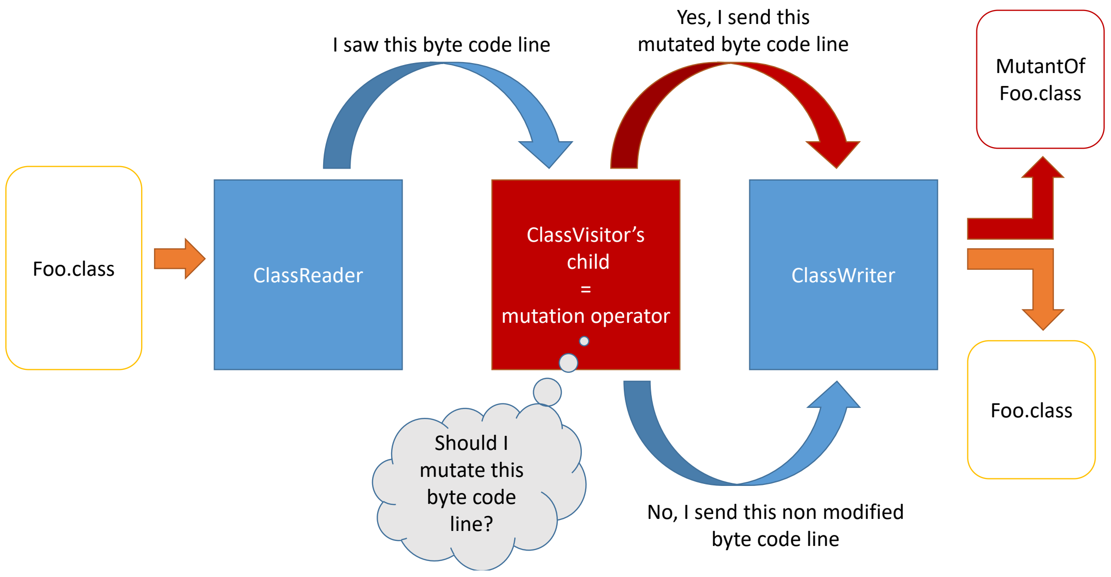
2. Then, create a specific `ClassVisitor` named `ClassWriter` that aims at writing anything it visits in a byte-code array. Finally, connect the `ClassWriter` to the `ClassVisitor`.

Figure 4.3: A chain of `ClassVisitor`'s reading `Foo.class` bytecode.



This gives us a chain of `ClassVisitors` and anything being read by the `ClassReader` is writen by the `ClassWriter`. Should we modify the `ClassVisitor` lying between the `ClassReader` and the `ClassWriter`, we could make the `ClassWriter` write alterated versions of the class being read. The default behavior of a `ClassVisitor` is to send anything it visits to the `ClassWriter`, as we explained before. Hence, it calls the `visit` method of the `ClassReader` after visiting himself the code that the `ClassReader` orders him to visit. By creating a child of a `ClassVisitor` and by overriding one of the inherited `visit` methods, one can create a stop point in the lecture of the class that is triggered by the overidden `visit` method. Thus, this stop point is triggered by a specific byte-code instruction being read. In the core of the overriden `visit` method, a programmer may define additionnal byte-code instructions to send to the `ClassWriter`. Or, a programmer may also suppress the call to the `ClassWriter` in order to delete instructions that the `ClassWriter` should write. Henry Coles [3] exploits this idea by extending a `ClassVisitor` class for each mutation operator he created. Hence, in the core of the overriden methods, he states the byte-code to be added or to be removed, his custom class forming a ready-to-go mutation operator (see figure 4.4).

Figure 4.4: A chain of `ClassVisitor`'s reading `Foo.class` bytecode and mutating it when reading specific byte code lines.


I saw this byte code line
Yes, I send this mutated byte code line
MutantOf Foo.class
Foo.class
ClassReader
ClassVisitor's child = mutation operator
ClassWriter
Should I mutate this byte code line?
No, I send this non modified byte code line
Foo.class

Moreover, the `ClassVisitor`'s lying between the `ClassReader` and the `ClassWriter` are very specific in PIT as those are `MethodVisitor`'s. A `MethodVisitor` is a class that works in the same way as `ClassVisitor`'s, but is specifically designed to read byte-code instructions lying in methods. Therefore, PIT's mutation operators are restrained to modifications of instructions lying inside methods of the class-under-test.

Although ASM is very powerful, it was engineered focusing on performance. Therefore, the library is usually used by performing a single reading of a class and triggering the modifications on the go. This has a serious impact on the mutation operators' implementation possibilities. Hence, one cannot perform very complex changes using the library. For instance, the removal of a code pattern conditioned by another code pattern that must appear later in the class under lecture is impossible to describe.

In the next sub-sections, we state the implementation of each of our mutation operators. Note that in 4.2, for each abstract mutation operator, we supplied one specification. However, this specification could be satisfied in many ways. Hence, for instance, there are many ways to use Message Digests in JAVA but we limited ourselves to the standard `java.util.Digest` to replace secured digests with MD5 (UWMD). In our research, we usually limited ourselves to one implementation supplied for each specification but many others should be considered for the future work.

Replacing secured objects with unsecured

UPPRNG: Use Predictable Pseudo Random Number Generator

The mutation operator replaces any call of `secureRandom.nextBytes(byteArray)` by `(new Random).nextBytes(byteArray)` where `secureRandom` is an instanciated `SecureRandom` object.
The use of `Random` instead of `SecureRandom` injects a predictable pseudo random number generator use. This mutation operator can therefore lead to weaknesses in secure communications or encryption. Thus, it facilitates the work of the attacker.

UWMD: Use Weak Message Digest

This mutation operator replaces any call to `new MessageDigest("X")` constructor by `new MessageDigest("MD5")`. The `java.security.MessageDigest` class provides the functionality of a message digest algorithm, such as SHA-1, SHA-256 or MD5. One specifies the desired algorithm with the `String X` passed to the constructor. In the implementation of our operator, X represents any string. Therefore, the operator may lead to equivalent mutants when X is already MD5. This operator can introduce weaknesses in hashes because of high collision inherent to the MD5 hashing function.

REIS: Remove Encryption In Socket

Operator that replaces an instance of `(Socket) SSLSocketFactory.getDefault().createSocket("address", portNumber)` by `(Socket) SocketFactory.getDefault().createSocket("address", 80)`. This operator will create a `Socket` that uses HTTP instead of HTTPS. It can lead to unencrypted communications that can be read by an attacker intercepting the network traffic.

PSQLI: Permit Sql Injection

Mutation operator that replaces the initialization and the execution of a `PreparedStatement` by the initialization and execution of a `Statement`, thus enabling SQL injection.

For the operator to be applied, the following pattern must be found in one method:

```
PreparedStatement preparedStatement =
    connection.prepareStatement("querryContainingOneOrSeveral'?'");
preparedStatement.setX1(1, value_1);
.
.
.
preparedStatement.setXn(n, value_n);
// where Xi are in {Float, Double, Boolean, Long, String, Int}

preparedStatement.execute() or executeQuerry() or executeUpdate();
```

The operator will only change the last line of the pattern to `connection.createStatement().execute("querry where '?' are replaced by the values set before")`.

Hence, the operator keeps the assignments of each input value in the `PreparedStatement` *e.g* when it recognizes a `updateSales.setString` or a `updateSales.setInt`, etc.

For instance,

```
PreparedStatement updateSales = conn.prepareStatement("update COFFEES set SALES = ?
    where COF_NAME = ?");
updateSales.setInt(1, nbSales);
updateSales.setString(2, coffeeName);
updateSales.execute();
```

will be mutated to

```
PreparedStatement updateSales = conn.prepareStatement("update COFFEES set SALES = ?
    where COF_NAME = ?");
updateSales.setInt(1, nbSales);
updateSales.setString(2, coffeeName);
conn.createStatement().execute("update COFFEES set SALES = '"+nbSales+"' where
    COF_NAME = '"+coffeeName+"'");
```

This mutation operator can lead to SQL injections because of the security a `PreparedStatement` proposes and that a `Statement` does not support. Hence, a `PreparedStatement` sends the precomputed query without its inputs (`coffeeName` and `nbSales`) to the DBMS. Then, the DBMS waits for the inputs that are sent in a second step in order to finalize the construction of the query and execute it. In a `Statement`, the query is built with its inputs before sending it as a `String` to the DBMS. Thus, an attacker may insert SQL code in the inputs in order to modify the result of the query. For instance, giving the following character sequence `coffeeName = "randomName' or 'a'='a";` for the `coffeeName` input will update all the `SALES` of the `COFFEES`.

Removing security features on JAVA objects

XMLPVXXE : Make XML parser vulnerable to XML eXternal Entity attack

This operator has two different implementations. Hence, there are several ways to create XML parsers'. In our research, we considered two different implementations for this operator. One implementation removes a DDOS sanitizing feature on `org.xml.sax.XMLReader`, the second achieves the same operation on a `javax.xml.parsers.SAXParserFactory` that is used to create a `javax.xml.parsers.SAXParser`.

XMLReader's implementation : The mutation operator injects `XMLReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", false);` before any call of `XMLReader.parse(InputSource)`. This operator will make the `XMLReader` vulnerable to XXE attacks if it parses input from an external source.

SAXParser's implementation : The mutation operator injects `saxParserFactory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", false);` before any call of `saxParserFactory.newSAXParser()`.
This operator will make any `SAXParser` created by `saxParserFactory` vulnerable to XXE attacks.

XMLPVXEE : Make XML parser vulnerable to XML eXternal Entity Expansion attack

Similarly to XMLPXXE, this operator has two different implementations.

XMLReader's implementation : The mutation operator injects `xmlReader.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false);` before any call of `xmlReader.parse(InputSource);`.
The call of `xmlReader.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false);` will disable the checking for DOS attacks before parsing. Therefore, the operator will make an `XMLReader` vulnerable to DOS attacks.

SAXParser's implementation : The mutation operator injects `saxParserFactory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false);` before any call of `saxParserFactory.newSAXParser()`.
The call of `saxParserFactory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false)` will disable the checking for DOS attacks before the `SAXParser` created parses. This mutation operator will make any `SAXParser` created by the `saxParserFactory` vulnerable to DOS attacks.

UC : Unsecure Cookie

This operator specifically aims at creating unsecured cookies in a server's code. The operator removes any call of `cookie.setSecure(true);` where`cookie` is a `javax.servlet.http.Cookie`. The instruction `cookie.setSecure(true)` orders to add a flag to the cookie disallowing a browser that receives it to send it back in an insecure communication. This mutation operator can thus enable attackers to read private information stocked in cookies, if they intercept the communication.

RHTTPOFC : Remove HTTP-Only Flag from Cookie

This operator specifically aims at creating unsecured cookies in a server's code. This mutation operator removes any call of `cookie.setHttpOnly(true);` where `cookie` is a `javax.servlet.http.Cookie`. The method `cookie.setHttpOnly(true);` adds a flag to the cookie sent by the server to the browser, ordering the browser to make sure that the cookie can not be read by a malicious script. This mutation operator can lead to session hijackings using cross-site scripting.

Weakening encryption algorithms

URSAWSK : Use RSA With Short Key

This mutation operator makes the program use a small key for RSA encryption (512bits) where a secured-size (>=2048bits) key was used.
First, the operator mutates the initialization of the key. Moreover, it replaces any call to `keyPairGenerator.initialize(X);` where `X>=2048` by `keyPairGenerator.initialize(512);`. Here, `keyPairGenerator` is a `java.security.KeyPairGenerator`.

In a `MethodVisitor`, the only way to know that the `KeyPairGenerator` is for RSA-use is to have the creation of the `KeyPairGenerator` in the same method as its initialization. Hence, `KeyPairGenerator`'s declare the algorithm they are used for at construction, *e.g* by using `keyPairGenerator = KeyPairGenerator.getInstance("RSA");`. Therefore, the mutation operator is applied only if it can find the construction `keyPairGenerator = KeyPairGenerator.getInstance("RSA")` in the same method and before the initialization.
As explained before, this mutation operator can lead to weaknesses in RSA-secured communications. For instance, it facilitates a brute force attack.

UBFWSK : Use Blowfish With Short Key

This mutation operator makes the program use a small key for BLOWFISH encryption (64bits) where a secured-size key (>=128bits) was used.
It replaces an initialization `keyGenerator.init(X);` where `X>=128` by `keyGenerator.init(64)`.
In a `MethodVisitor`, the only way to know that the KeyGenerator is for BLOWFISH-use is to have the construction of the `javax.crypto.KeyGenerator` in the same method as its initialization. Hence, `KeyGenerator`'s declare the algorithm they are used for at construction, *e.g* by using `keyGenerator = KeyGenerator.getInstance("BLOWFISH");`.
Therefore ,the mutation operator is applied only if it can find the construction `keyGenerator = KeyGenerator.getInstance("BLOWFISH");` in the same method and before its initialization.
As stated before, this operator can lead to weaknesses in BLOWFISH-secured communications,

it facilitates a brute force attack for instance.

Removing standard sanitization functions

RPTS : Remove Path Traversal Sanitization

The operator removes any call to the user-input-sanitization method `org.apache.commons.io.FilenameUtils.getName(name);`.
Hence, `FilenameUtils.getName(name)` method works in the following way:
if `name = "a/b/c.txt"`,
`FilenameUtils.getName(name)` will return `"c.txt"`.
The sanitization method prevents an attacker from accessing to the folders inside the one to which a simple user has access. This mutation operator can therefore lead to path traversal vulnerabilities.

RRS : Remove Regex Sanitization

For this mutation operator, we provide two implementations. The first mutates `Pattern`'s, the second mutates calls to the `String.matches(String regex)` method.

Implementation using Patterns : One way to use regex's in JAVA is to use the `java.util.regex.Pattern` class. It is used in the following way:

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches(); // b contains true
```

The `Pattern` defines the regular expression to be matched by using Pattern.compile(String regex). Then, a matcher is created by using p.matcher(CharSequence input), the input containing a `CharSequence` that should be verified against the regex. Finally, a `boolean` stocks the check that the `CharSequence` matches the `regex`.

We designed a mutation operator that replaces any call to `p.matches(regex)` by `p.matches(dummyRegex)`. The regex `dummyRegex` is designed to let anything pass except from one rare unicode character.
The result is that the mutation operator can suppress input sanitization and facilitate an attack.

Implementation using String.matches method : Another way to use regex's in JAVA is to use the `java.lang.String.matches(String regex)` method. It is used in the following way:

```
String expression = "aaaaaab";
boolean b = expression.matches("a*b"); // b contains true
```

The input to verify is stocked in `expression`. Then, the method `matches` is used to check whether the `expression` matches a regex.

We design a mutation operator that replaces any call of `String.matches("aProperRegex")` by `String.matches("dummyRegex")`. The dummy regex is the same as in the previous implementation.
The result is thus the same as in the previous implementation, the mutation operator can suppress input sanitization and therefore, facilitate an attack.

Replacing strong with weak encryption algorithms:

UDESISE : Use DES In Symmetric Encryption

The mutation operator's implementation replaces the creation and initialization of a secure symmetric encryption object (`javax.crypto.Cipher`) by the creation and initialization of the object, modified to use *Data Encryption Standard (DES)*. Hence, `javax.crypto.Cipher` may be used in the following way:

```
// creation of the Cipher
Cipher c = Cipher.getInstance("SECUREALGORITHM/MODE/PADDING");

// initialization of the Cipher
c.init(Cipher.ENCRYPT_MODE or Cipher.DECRYPT_MODE, key);

// use of the Cipher
c.doFinal(...) or c.update(...);
```

First, the `Cipher` is created by using `getInstance` method. `getInstance` takes as a `String` parameter the specification of algorithm that the `Cipher` should encompass. When this method is called, the *Security Provider* of the application's *JAVA runtime environment* is called to provide a correct implementation of the specified encryption algorithm. We do not aim in this work at explaining how these mechanisms work in *JAVA* but we rather refer to the interested reader to a JAVA security guide[5]. Next, the `Cipher` is *initialized* for encryption or decryption by using `init(Cipher.ENCRYPT_MODE,key)` or `init(Cipher.DECRYPT_MODE,key)`, respectively. Finally,

[5]ORACLE provides a guide that documents how the security providers are used in JAVA on the following web-page http://docs.oracle.com/javase/7/docs/technotes/guides/security/overview/jsoverview.html

the `Cipher` is used to encrypt or decrypt text by using `doFinal(...)` or `update(...)`. We do not give the parameters of the last two methods as their different implementations are numerous.

As all our mutation operators are `MethodVisitor`'s, the following pattern that is found in the core of a method triggers the mutation:

```
// creation of the Cipher
Cipher c = Cipher.getInstance("SECUREALGORITHM/MODE/PADDING");

// initialization of the Cipher
c.init(Cipher.ENCRYPT\_MODE or Cipher.DECRYPT\_MODE, key);
```

Thus, the operator mutates both lines by the following code:

```
// creation of a DES/ECB/PKCS5Padding Cipher
Cipher c = Cipher.getInstance("DES/ECB/PKCS5Padding");

// initialization of the Cipher with a DES specific key that is 64 bits long
c.init(Cipher.ENCRYPT_MODE,new SecretKeySpec(key.getEncoded()\allowbreak,0,8,"DES"));
```

The method `key.getEncoded()` returns the byte array of the `key`. The call to `new SecretKeySpec (key.getEncoded(),0,8,"DES")` creates a `javax.crypto.spec.SecretKeySpec`, a key specified for `DES` use. The `SecretKeySpec` is created by truncating the previously used key to 64 bits, a standard size for a DES key. Thus, the `Cipher` is mutated to encrypt or decrypt `DES` text's and its initialization is in accordance with its usage. Hence, if we do not mutate the second line, the usage of the `Cipher` could throw an `Exception` at run-time, because the key in parameter could have a inappropriate size for `DES` use.

Because DES is known to be insecure, this mutation operator can lead to weaknesses in symmetric encryption.

Unfortunately, we could not find a way to implement this operator correctly in `PIT` without modifying significantly the tool's implementation. As such a modification could lead to errors in other parts of the tool, we tested our implementation's draft in another project and it gave us promising results. Therefore, we left the operator in the core of this thesis, as it highlights the issue in `PIT` and to introduce its specification in the security-aware mutation operators.

UECBISE : Use ECB In Symmetric Encryption

Again, this mutation operator mutates `Cipher` objects. Hence, it replaces any initialization of a `Cipher`, `Cipher.getInstance("X/Y/Z")` where `Y` is not `"ECB"` by `Cipher.getInstance("X/ECB/Z")`. If `Y` and `Z` are empty, the operator replaces the initialization by `Cipher.getInstance("X/ECB/PKCS5Padding)`.
The ECB mode means that if two plain-text blocks are identical, the obtained cipher-text blocks will also be identical. The padding is a default padding.
This mutation operator can thus lead to weaknesses in encryption. Therefore, it facilitates the decryption of a ciphered-text by an attacker by introducind a pattern redundancy in the encrypted messages.

Removing authentication mechanisms:

RHNV : Remove Host Name Verification

This mutation operator replaces any verification of a HostName using `(boolean) HostnameVerifier.verify(String hostname, SSLSession session)` by `true`.
Hence, to use `javax.net.ssl.HostNameVerifier.verify` is a standard way to "verify that the host name is an acceptable match with the server's authentication scheme" [46].
`HostNameVerifier` is an `Interface` and, usually, the method `HostNameVerifier.verify`'s implementation verifies the certificate of the host specified by its host-name in the `SSLSession` before returning true or false.
This operator can thus lead to vulnerabilities in the authentication process of a program since the mutant accepts any host.

Chapter 5

Experimentation & Discussion

In this chapter, we describe our experimentation in section 5.1 and discuss the results in section 5.2.

5.1 Experimentation

In this section, we first derive the problem discussed in chapter 3 into four formal research questions. Then, we state the experimentation process we followed to answer these questions in section 5.1.2. Finally, in the sub-section 5.1.3, we present the case studies we experimented on.

5.1.1 Deriving research questions from the problem

In chapter 3, we stated the following problems :

1. Non security-aware mutation operators are unlikely to incidentally generate vulnerable mutants.
2. Mutation operators specialized in introducing vulnerabilities at code level were never experimented before.

In this thesis, we aim at resolving these problems. Therefore, we verify the relevance of a new set of security-aware mutation operators, specifically tuned for introducing vulnerabilities. For demonstrability, we focus on the JAVA language, which is very popular in web development and other security-sensible areas of development. Nevertheless, the first step is to validate our intuition, check that standard mutation operators are indeed unlikely to generate vulnerable mutants. Here, we chose to focus on the mutation operators implemented in a very popular mutation tool named *PITest*. This interrogation leads us to state the first research question (RQ):

(RQ1) Are the standard operators of PIT likely to introduce vulnerabilities?

Next, we explore potential operators, inspiring ourselves on the expertise of a security static analyzer (*Findbugs-sec*). Then, we implement those operators in *PITest*. The next step is to verify our operators' ability to introduce real vulnerabilities in real projects. This leads us to state our second research question:

(RQ2) Does our mutation operators introduce detectable vulnerabilities?

Then, as one of the major problems in mutation testing is the number of mutants generated but is also the main guarantee of generating quality test-suites, we explore the prevalence of our mutation operators in open source projects. This indeed aims at assessing operators applicability and can help prioritizing their application. Resuming our idea leads us to the RQ3.

(RQ3) How prevalent are these types of faults on open source projects?

Finally, we explore a last but not the less important question. What if the vulnerabilities our operators introduce were actually redundant with the faults standard mutation operators introduce? This fact could limit our operators' value compared to the standard ones. Therefore, we investigate this potential threat in our last research question, RQ4.

(RQ4) Are the vulnerabilities introduced easily killed by fault oriented test suites?

In the next section, we detail the experimentation process that we followed in order to answer our four research questions.

5.1.2 Experimentation process

During our experimentation, we first began by investigating the possibility of creating the security-aware mutation operators. After the exploration of the state of the art summarized in chapter 2, we took inspiration on security static analyzers, tools capable of highlighting vulnerable patterns in the code of an application (see chapter 2). Indeed, if we are provided with both a vulnerability pattern at the code level and its fix, we are able to design a mutation operator introducing the vulnerable code. In this thesis, we focused on the vulnerable code patterns that *FindBugs-sec* was able to identify and reverted those manually in order to construct our mutation operators that introduce the vulnerable code patterns.

Then, we implemented those operators in the *PITest* mutation tool. We detailed our operators definition and implementation in chapter 4.

Next, we wanted to verify our operators and in a reproducible and automatic way. Also, we wanted to evaluate our operators' usage in real world applications. Therefore, we found four open-source projects, two web applications and two bit-torrent clients (applications that should be concerned with security) and generated their mutants according to our mutation operators. More details about the case studies will be given in the next section 5.1.3. Then, we launched FindBugs on each mutated class and on its mutants to generate FinbBugs standard reports. The final step was to compare the report of the class to the reports of its mutant in order to verify the correct introduction of the vulnerabilities in the mutants. This part of the process gave us the opportunity to answer to *RQ2* and *RQ3*.

Using the same automated process but using PIT's original set of operators to generate the mutants, we could answer *RQ1*. Hence, we generated the mutants of one of our open source applications then assessed the number of vulnerabilities introduced in the application using FindBugs, again.

Finally, we focused on our last research question. We could find a high coverage test suite for one of our open source web applications. By using PIT mutation tool restricted to our mutation operators and allowing it to run this test suite, we could answer *RQ4*.

5.1.3 Case studies

iTrust

iTrust[1] is a web application developed and maintained by the students of NCState University and consists of 24,785 lines of code. It provides a platform accessible to patients and doctors, to keep track of the patient's medical history. The project available to the public insuring its accessibility by providing instructions for its installation[2].

Vuze Azureus

Vuze[3] is a popular open-source Bittorrent client, consisting of 186,247 lines of code.

[1]`https://sourceforge.net/projects/itrust/` (version 21.0.01)
[2]`http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=home_deployment_instructions`
[3]`https://sourceforge.net/projects/azureus/` (version 5.7.40)

Open-Legislation

OpenLegislation[4] is an open source web application developed and maintained by the New York State Senate. The goal of this application is to give access to several NYS's data including bills, resolutions and laws. It consists of 912 classes and is 33,819 lines of code

Ants Peer-to-Peer

Ant's Peer-to-Peer is an open-source Bittorrent client[5] (consisting of 19,399 lines of code), like Vuze.

5.2 Discussion

This section discusses the results we obtained by following the process detailed in section 5.1.2. In order to organize the discussion, we go through the results by answering every research questions that we stated in section 5.1.1.

5.2.1 RQ1: Are PIT's default operators likely to produce vulnerabilities?

For the exploration of this research question, we generated iTrust's mutants using PIT, restricted to its default set of mutation operators. Then, we ran FindBugs coupled to its sec-plugin to investigate the presence of vulnerabilities in the mutants. We specifically tuned FindBugs to check every potential issue, even those of low confidence, at the cost of performance. Because of the high number of mutants generated for iTrust's project (more than 44000) and the time required to analyze one mutant (on average two minutes), we restricted the set of classes to mutate to 33 classes.

As shown in table 5.1, PIT generated a set of 5486 mutants containing only two vulnerabilities. Both vulnerabilities were related to SQL injection, and were detected on a mutant where the mutation removed a method call and on another mutant where the mutation removed a conditional execution. Although PIT's standard set of operators was indeed able to introduce vulnerabilities, their number appears to be low considering the required generation and run time of the mutants. We therefore conclude that PIT's standard mutation operator set doesn't suit security test suites evaluation, the efforts required to generate and evaluate the mutants largely outweigh the meagre benefits in terms of introduced vulnerabilities.

[4]`https://github.com/nysenate/OpenLegislation` (version 2.2)
[5]`https://sourceforge.net/projects/antsp2p/` (version beta1.6.0)

Table 5.1: We mutated iTrust with PIT's standard mutation operator set. The table records the number of mutants and vulnerabilities that were generated for each operator. The vulnerabilities' presence relays on FindBugs and its security plugin.

Operator name	*#mutants*	*#Vulnerabilities*
ArgumentPropagationMutator	42	0
ConditionalsBoundaryMutator	48	0
ConstructorCallMutator	431	0
IncrementsMutator	12	0
InlineConstantMutator	696	0
MathMutator	41	0
MemberVariableMutator	83	0
NegateConditionalsMutator	368	0
NonVoidMethodCallMutator	1539	1
RemoveConditionalMutator_EQUAL_ELSE	320	0
RemoveConditionalMutator_EQUAL_IF	320	0
RemoveConditionalMutator_ORDER_ELSE	48	0
RemoveConditionalMutator_ORDER_IF	48	1
RemoveIncrementsMutator	12	0
RemoveSwitchMutator	15	0
ReturnValsMutator	289	0
SwitchMutator	2	0
VoidMethodCallMutator	1172	0
Total	5486	2

5.2.2 RQ2: Are our mutation operators likely to produce vulnerabilities?

Now that we know that PIT's standard set of mutation operators is not suitable for security testing, our next research question investigates the extend to which our set of operators can do better, *i.e* introduce more vulnerabilities. The first step required by this task is to verify our mutations operators' capability in introducing vulnerabilities.

To assess this problem, we first used the same generation and checking technique as in 5.2.1: we generated the mutants using PIT and evaluated the number of vulnerabilities introduced with FindBugs. PIT's engine was this time restricted to our set of mutation operators. To perform the experiment, we manually created a sample project containing several classes, each one implemented to trigger one specific mutation, accordingly to our set of security aware operators.

Due to the possibility of false positives inherent to static analyzers, we also manually implemented a security test suite able to kill every (non equivalent) mutant we introduce. Overall, we exploited an hybrid analysis composed of a static and a dynamic analysis to verify the introduction of vulnerabilities. The static and dynamic analysis combined can be seen as a sanity check. The dynamic analysis also assessed the fact that the mutation operators introduced useable (killable) mutants.

Static analysis:

Table 5.2 reports FindBugs' results on our sample project. `RRS` and `UDESISE` were not evaluated. We chose not to evaluate `RRS` because it was not inspired by a FindBugs pattern, and could therefore not be found by FindBugs. Regarding `UDESISE`, the operator's implementation is currently experimental in PIT, as the tool wasn't designed to perform higher order mutation.

Overall, we can see that 9/13 (nearly 70%) operators generated a vulnerability that could be found by FindBugs. We list the non-recognition causes of the 30% remaining.

- **Remove Certificate Verification (RCV)** The vulnerability is actually well introduced. Hence, `HostNameVerifier.verify(...)`'s result was replaced by `true` in the mutant. The vulnerability was not recognized because FindBugs identified a vulnerable HostNameVerifier based on the HostNameVerifier's code, not it's usage. Indeed, for FindBugs, a vulnerable `HostNameVerifier` is identified by a constant `return true;` in all its `.verify(...)` methods.

- **Xml Parser Vunerable to XXE/XEE (XMLPVXXE/XMLPVXEE)** The vulnerability is also well introduced because the operator removed a secure parsing feature on an XML reader. However, the secure original program contained the line:

```
XMLReader.setFeature(SecurizingFeature, true)
```

  preceding a parsing. Thus, the mutant contained the following code:

```
XMLReader.setFeature(SecurizingFeature, true);
XMLReader.setFeature(SecurizingFeature, false);
XMLReader.parse(input);
```

  Our hypothesis is that FindBugs analysed the first line to directly return that the vulnerability was handled by the programmer.

- **Remove Encryption In Socket (REIS)** The manual verification of the vulnerability's introduction led us to conclude that FindBugs' identification pattern is limited to the following one:

```
Socket s = new Socket(address,port);
```

The problem is that we introduced an equivalent code using a different pattern.

```
Socket s = SocketFactory.getDefault().createSocket(address,port);
```

We assume that the two manners of creating an unsecured socket are equivalent after inspection of `createSocket(...)` method's specification.

Table 5.2: Injected classes of vulnerabilities that were identified by FindBugs (with the security plugin), in a sample project. We mutated this project and verified the presence of vulnerabilities by comparing the static analysis reports of the mutants and the original programs. YES signifies that the injected vulnerability was identified by FindBugs.

	UPPNRG	*RPTSF*	*UWMD*	*RCV*	*XMLPVXXE*
Recognized?	YES	YES	YES	NO	NO
	XMLPVXEE	*REIS*	*UC*	*RHTTPOFC*	*URSAWSK*
Recognized?	NO	NO	YES	YES	YES
	UBFWSK	*PSQLI*	*UDESISE*	*UECBISE*	*RRS*
Recognized?	YES	YES	/	YES	/

Dynamic analysis:

After designing and writing our test suite, we used PIT's entire mutation tool, to generate the mutants with respect to our operators and ran those against a custom-made security test suite. The table 5.3 subsumes our results.

First, it is noteworthy that all the analysis-considered mutants were killed by the security test suite. This is a very good result as we dreaded the difficulty of finding the introduced security faults. We prove that killing the mutants introduced is actually feasible. Therefore, this experience insures that the mutants introduced are actually usable in the improvement process of a test-suite.

Table 5.3: Injected classes of vulnerabilities that were identified by a custom-made security test-suite in a sample project. We mutated this project and verified the presence of vulnerabilities by running a custom-made security test-suite against the mutants. YES signifies that the injected vulnerability was identified by the suite.

	UPPNRG	*RPTSF*	*UWMD*	*RCV*	*XMLPVXXE*
Recognized?	YES	YES	YES	YES	YES
	XMLPVXEE	*REIS*	*UC*	*RHTTPOFC*	*URSAWSK*
Recognized?	YES	YES	YES	YES	YES
	UBFWSK	*PSQLI*	*UDESISE*	*UECBISE*	*RRS*
Recognized?	YES	YES	/	YES	YES

Although it is not reported in the tables, the operators also generated (living) equivalent mutants. However, this fact does not affect the results since equivalent mutants are a problem inherent to mutation testing.

Conclusion:

With regards to the previous sub sections' results, we are now able to answer our second research question. We show that we were able to introduce vulnerabilities with our mutation operators with a double check, a static analysis that could find 70% of the vulnerabilities and a dynamic analysis that could identify all the generated vulnerabilities. The dynamic analysis eliminates the static analyzer's false positives and eliminates its false negatives for `REIS, XMLPVXEE, XMLPVXEE and RCV`. The false negatives either come from the static analyzer's incompleteness `(REIS)` or optimisations `(XMLPVXXE/XMLPVXEE)`. Although our mutation operators were inspired by FindBugs patterns, the vulnerabilities they produced were not all found by the tool. Therefore, it suggests that our operators could have another applicability in static analysis tools' validation.

5.2.3 RQ3: How prevalent are the vulnerabilities in open source projects?

The last research question responds partly to the effectiveness comparison between our mutation operators and PIT's standard set in the goal of generating vulnerabilities. After answering RQ1 and RQ2, we state that PIT's set of operator is not suitable for security testing improvement and that our mutation operators introduce real vulnerabilities. To end with this point, we now would like to assess the vulnerabilities' prevalence in real projects.

In the next experiment, we use PIT, restricted to our mutation operators, on four open source projects. Table 5.4 displays the results obtained.

Table 5.4: We mutated open source projects with new operators using PIT. The purpose was to assess the prevalence of security-oriented mutants.

	iTrust	*VUZE*	*OPENLEGISLATION*	*ANTSP2P*
#classes in input	405	4669	912	406
#lines in input	24,785	186,247	33,819	19,399
#classes mutated	33	30	57	9
#mutants generated	62	57	154	18

First, we notice that the number of vulnerabilities introduced does not depend on the size of the considered project. Hence, iTrust and Vuze have a comparable number of security oriented mutants whereas their size is significantly different. This is an interesting result since traditional mutation operators are known to exponentially generate mutants while increasing the size of the projects on which they are applied. The fact that we do not have the same behavior highlights the nature of our security-aware mutation operators. Hence, our operators aim at testing security, which is a non-functional requirement whereas traditional mutation operators aim at testing functional requirements. Therefore, the number of mutants that will be generated using our operators will increase only if the security of the SUT increases, disregarding all the functional increments that could be made to the SUT.

While we used manual code review, we also noted that since iTrust followed the same design for a large amount of its database interaction, the feature-corresponding mutation pattern (*PSQLI*) was triggered in a significant proportion. This is an interesting result if it is generalizable. Hence, if developers tend to develop pieces of code having a similar purpose in a similar way, the tester can be confident to have tested a large amount of security feature instances in the project under test by using our mutation operators.

More generally, we observe the relative generated vulnerabilities' prevalence. Hence, vulnerabilities are hard to find in real world projects, giving the tester too few examples to get experience with. Having these amounts, 62, 57, 154 and 18 of vulnerabilities available in an acceptable time (the generation never took more than 10 minutes) can be estimated valuable for security testers.

To compare the prevalence of vulnerabilities generated with our mutation operators to PIT's standard set, let us state that every vulnerability we generate was detected by FindBugs, whereas it took the generation of more than 5000 mutants to generate two vulnerabilities for iTrust, using PIT's standard operators (see *RQ2*). Therefore, we are confident for the applicability and the scalability of the security aware operators we propose, in comparison to what was available before.

Table 5.5 details the mutation operators generating the set of mutants which we just have been discussing for each open source project.

Table 5.5: Number of security-aware mutants generated on four open source projects. The table entries record the number of mutants generated per mutation operator and project. Non referenced operators did not produce any mutants.

	UPPNRG	*UWMD*	*PSQLI*	*UECBISE*	*RRS*	*TOTAL*
ITRUST	1	0	39	0	22	62
VUZE	8	2	0	9	38	57
OPENLEGISLATION	0	0	0	0	154	154
ANTSP2P	2	4	0	11	1	18

First, we can observe that there are disparities in the mutants' frequencies, classified by the mutation operator applied. Hence, `RSS` seems to find applications easily while `UPPNRG` hardly finds an application in every project. We also note that only four mutation operators were useful in four open source projects' review. However, as `PSQLI` demonstrates, an operator can be useless in several projects and highly demonstrate its value in another.

Yet, we assume that `RSS`'s "popularity" is also understandable regarding its frequency in web applications. A regex sanitization, is understandably more common than a Blowfish usage, for instance.

We therefore conclude that many security-aware faults appear in the selected projects. Though, we note that further case studies are required to certify these first results on the prevalence of our operators.

5.2.4 RQ4: Ease of killing vulnerabilities introduced with fault-oriented test suites

The previous research questions assess the usability of the new mutation operators in order to improve security test suites. As we know, vulnerabilities form a subset of the possible faults in a program. If existing fault-oriented test suites can kill the mutants generated accordingly to security operators, there is no real need to use these operators.

To assess the research question following this interrogation, we used iTrust's test suite composed of more than 2200 tests cases and ran those (using PIT) against the 62 mutants we generated, accordingly to our mutation operators. The mutation coverage report is detailed in the following table.

Table 5.6: We ran 62 mutants generated using our mutation operators on iTrust against 2254 (fault finding oriented) test cases available in iTrust. We achieved this using PIT and this table details the mutation coverage report resulting the experiment.

Mutation operator	*number of mutants*
no coverage (total)	*5*
RRS	1
PSQLI	4
survived (total)	*44*
RRS	8
PSQLI	35
UPPNRG	1
killed (total)	*13*
RRS	13
Total	62

A first observation is that a minority of mutants where killed (≈21%). Although the experiment should be repeated on other projects to assess the result's representativeness, we believe it indicates that fault-oriented test suites cannot find easily the vulnerabilities we introduce. Therefore, we conclude that the set of mutation operators we propose suits its purpose to introduce new types of faults, killable by specialized (security) test suites only.

We remark that, after manual reviewing the killed mutants, we found that those were killed by regex testing suites. However, the suites did not seem to follow defined rules of design, and did not appear being desingned to assess application's security.

5.2.5 Threats to validity

Internal validity

Our conclusions may be threatened by the used tools. The mutation operators we proposed were implemented in PIT mutation tool, in which all the mutation operators rely on bytecode manipulation. Hence, PIT articulates the ASM JAVA bytecode framework as an abstraction layer for the mutant generation, forcing the developer to follow this line while designing their new mutation operators. The byte code manipulation being an extra difficulty to manage while implementing the mutation operators, potential defects might influence our results. However, we used code review on the operators' implementation and manually inspected the generated mutants during every experiment. Furthermore, the static and dynamic analysis we performed in *RQ2* on the generated mutants also reassures us about the correct implementation of our operators. Hence, in *RQ2*, FindBugs could identify 70% of the vulnerabilities generated and a custom-made security test-suite was able to kill all the (non equivalent) mutants.

Construct validity

In *RQ1*, we choose iTrust to perform the experiment because of the high diversity of security concerns that the application should consider, inherently to its web application nature. This feeling is also confirmed in table 5.4 as a higher percentage of classes (8%) are mutated in iTrust, compared to the other considered projects. *RQ2*'s only purpose was to verify our operators' implementation. Following this idea, we mutated on average one custom-made class per operator, specifically implemented to trigger a corresponding mutation operator. We defend that this validation is sufficient for RQ2's goal. For *RQ3*, we performed our experimentation on 2 web applications and 2 Bit torrent clients. In future work, we aim at repeating the experiment, and increase the diversity and the number of projects to confirm the results we obtained. In *RQ4*, we chose iTrust because of the availability and the high coverage of its test cases (≈94%). The coverage was calculated using EclEmma's coverage plugin[6] for eclipse.

[6]`http://www.eclemma.org`

External validity

We took inspiration on patterns identified and documented by FindBugs sec plugin. It could be questionable whether the proposed patterns could be found in practice. Nevertheless, every pattern defined in the plugin relies on real vulnerability occurrences described in well-known reporting authorities (NIST, CVE), as stated in the plugin's documentation[7]. However, a high percentage of cyber-security incidents are caused by the exploitation of known vulnerabilities [47]. In our evaluations, we selected four open source projects that should be highly concerned by security issues, inherently to their nature (Web Applications and Bit torrent clients).

[7]`http://find-sec-bugs.github.io/bugs.htm`

Chapter 6

Conclusion & Perspectives

This thesis introduces security-aware mutation testing operators. Inspired by common vulnerability patterns, we have designed 15 new mutation operators for JAVA, which we implement in the mutation testing engine of PIT. We used FindBugs and its security plugin to assess whether standard PIT mutation operators are likely to introduce vulnerabilities, like those supported by our operators, and demonstrated that they fail to do so. Our case studies validated the purposes of our operators and revealed that certain types of vulnerabilities are prevalent in open source projects.

This work constitutes the first step towards a relatively new direction of mutation testing research which is the *mutation-based security testing*. Moreover, the major conclusions of the work we present in this thesis were published [20] in *Mutation2017*, an international workshop that aims at discussing new and emerging trends in mutation analysis. With the use of our mutants, security related test suites can be designed and documented. Other potential uses of our mutation operators are in education and in the systematic evaluation and comparison of fuzzing and other security testing tools. Overall, our goal is to use mutation to define adequacy criteria for security testing. Moreover, in the next lines, we state the potential benefits of our approach in the security testing techniques that we presented in chapter 2.

Code based techniques and static analysis: First, the potential of our technique may be easily highlighted in the education of security testers. Hence, our technique, in the way of the *Metasploit framework*[1], gathers a database of real-world vulnerabilities. Therefore, our mutation operators may be valuable in educating the security testers by providing potential vulnerability patterns but also by providing a correction for each of these patterns. Following

[1]Metasploit Pen Testing Tool is an open-source project that gathers information's on vulnerabilities that may be found in real-world projects. Doing so, it aims at helping penetration testing.

the same idea, the patterns we collect, considering future improvement, may form an effective database to include in the Code-Based security testing techniques.

Dynamic security testing tools and techniques: Next, the operators may also be used for generating vulnerabilities in real world projects. Here, the benefits we imagine lie in the evaluation of security tools and/or techniques. Hence, one could compare vulnerability scanners, fuzzers, or other dynamic security testing techniques by using our operators to generate vulnerable versions of a software and see which technique is capable of discovering the most vulnerabilities.

Regression security testing: Finally, we also consider the applicability of the new operators but also of the extended tool in the creation of security regression test suites. Hence, each operator illustrates the usage of a secured feature in the current code of the application and forces the tester to cover the case where the secured feature would be missing. Therefore, if one removes one of these secured features during the maintenance phase, the regression test suite will immediately alert the mistaking programmer. We strongly believe that this statement forms the major impact of our research as we could not find, to the best of our knowledge, another mutation based security testing technique covering this issue.

In the future, we plan to extend our work towards the following directions: First, we plan to consider a much larger set of security patterns that we will mine from open source projects. Second, we will assess our research questions on more subjects and confirm our observations with actual tests from fuzzing tools. Third, we would like thoroughly assess the practical benefits of our security testing metrics.

REFERENCES

[1] I. Gilchrist, "Software quality engineering: Testing, quality assurance, and quantifiable improvement. by jeff tian. published jointly by john wiley & sons, inc., hoboken, nj, u.s.a. and ieee computer society press, los alamitos, ca, u.s.a., 2005. isbn: 0-471-71345-7, pp 412: Book reviews," *Softw. Test. Verif. Reliab.*, vol. 16, no. 2, pp. 124–125, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1002/stvr.v16:2

[2] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[3] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 449–452.

[4] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Chapter one-security testing: A survey," *Advances in Computers*, vol. 101, pp. 1–51, 2016.

[5] G. McGraw, *Software security: building security in.* Addison-Wesley Professional, 2006, vol. 1.

[6] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh, "Technical guide to information security testing and assessment," *NIST Special Publication*, vol. 800, p. 115, 2008.

[7] C. P. . B. S. . B. F. ORBAC's authors, "Orbac: Organization based access control http://orbac.org/?page_id=21."

[8] M. Büchler, J. Oudinet, and A. Pretschner, "Semi-automatic security testing of web applications from a secure model," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, June 2012, pp. 253–262.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[10] P. Ammann and J. Offutt, *Introduction to software testing.* Cambridge University Press, 2008.

[11] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons, 1997.

[12] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, 2010, pp. 121–130.

[13] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 354–365.

[14] T. C. Thierry, M. Papadakis, Y. L. Traon, and M. Harman, "Empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *ICSE*, 2017.

[15] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *ISSTA*. ACM, 2014, pp. 315–326.

[16] S. Kim, J. A. Clark, and J. A. McDermid, "Class mutation: Mutation testing for object-oriented programs," in *Proceedings of the Net. ObjectDays Conference on Object-Oriented Software Systems.* Citeseer, 2000, pp. 9–12.

[17] C. Zhou and P. Frankl, "Mutation testing for java database applications," in *Software Testing Verification and Validation, 2009. ICST'09. International Conference on.* IEEE, 2009, pp. 396–405.

[18] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Integration testing using interface mutation," in *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on.* IEEE, 1996, pp. 112–121.

[19] P. Arteau, "Bug patterns - find security bugs http://find-sec-bugs.github.io/bugs.htm."

[20] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, and P. Heymans, "Towards security-aware mutation testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on.* IEEE, 2017, pp. 97–102.

[21] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," 1996.

[22] R. Geist, A. J. Offutt, and F. C. Harris, Jr., "Estimation and enhancement of real-time software reliability through mutation analysis," *IEEE Trans. Comput.*, vol. 41, no. 5, pp. 550–558, May 1992. [Online]. Available: http://dx.doi.org/10.1109/12.142681

[23] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, "Analysing and comparing the effectiveness of mutation testing tools: A manual study," in *International Working Conference on Source Code Analysis and Manipulation*, 2016.

[24] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on.* IEEE, 2009, pp. 220–229.

[25] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[26] T. Chekam, M. Papadakis, Y. Traon, and M. Harman, "Empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption." ICSE, 2017.

[27] M. Delahaye and L. du Bousquet, "A comparison of mutation analysis tools for java," in *Proceedings of the 2013 13th International Conference on Quality Software*, ser. QSIC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 187–195. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2013.47

[28] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: A mutation system for java," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 827–830. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134425

[29] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 433–436.

[30] T. Laurent, A. Ventresque, M. Papadakis, C. Henard, and Y. L. Traon, "Assessing and improving the mutation testing practice of pit," *arXiv preprint arXiv:1601.02351*, 2016.

[31] A. Garg, J. Curtis, and H. Halper, "Quantifying the financial impact of it security breaches," *Information Management & Computer Security*, vol. 11, no. 2, pp. 74–83, 2003.

[32] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[33] G. Tian-yang, S. Yin-Sheng, and F. You-yuan, "Research on software security testing," *World Academy of science, engineering and Technology*, vol. 70, pp. 647–651, 2010.

[34] M. Meucci, E. Keary, D. Cuthbert *et al.*, "Owasp testing guide v4," *OWASP Foundation*, vol. 4, 2015.

[35] M. Büchler, J. Oudinet, and A. Pretschner, "Semi-automatic security testing of web applications from a secure model," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on.* IEEE, 2012, pp. 253–262.

[36] R. Scandariato, J. Walden, and W. Joosen, "Static analysis versus penetration testing: A controlled experiment," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on.* IEEE, 2013, pp. 451–460.

[37] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[38] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using findbugs on production software," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 805–806. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297897

[39] T. Mouelhi, Y. L. Traon, and B. Baudry, "Mutation analysis for security tests qualification," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, Sept 2007, pp. 233–242.

[40] F. Autrel, F. Cuppens, N. Cuppens-Boulahia, and C. Coma, "Motorbac 2: a security policy tool," in *3rd Conference on Security in Network Architectures and Information Systems (SAR-SSI 2008), Loctudy, France*, 2008, pp. 273–288.

[41] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korosec, W. Krenn, R. Schlick, and B. V. Schmidt, "Model-based mutation testing of an industrial measurement device," in *Tests and Proofs*, ser. LNCS, vol. 8570. Springer, 2014, pp. 1–19.

[42] H. Shahriar and M. Zulkernine, "Mutation-based testing of buffer overflow vulnerabilities," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland*, 2008, pp. 979–984.

[43] A. K. Ghosh, T. O'Connor, and G. McGraw, "An automated approach for identifying potential vulnerabilities in software," in *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, 1998, pp. 104–114.

[44] F. Dadeau, P.-C. Héam, R. Kheddam, G. Maatoug, and M. Rusinowitch, "Model-based mutation testing from security protocols in hlpsl," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 684–711, 2015.

[45] J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke, "Mutation testing of memory-related operators," in *Software testing, verification and validation workshops (ICSTW), 2015 IEEE eighth international conference on.* IEEE, 2015, pp. 1–10.

[46] ORACLE, "Hostnameverifier (java platform se 7) https://docs.oracle.com/javase/7/docs/api/javax/net/ssl/HostnameVerifier.html."

[47] I. Schieferdecker, J. Grossmann, and M. Schneider, "Model-based security testing," *arXiv preprint arXiv:1202.6118*, 2012.

APPENDIX

Appendix A

Mutation 2017 Workshop Publication

Towards Security-aware Mutation Testing

Thomas Loise*†, Xavier Devroey*, Gilles Perrouin*, Mike Papadakis†, and Patrick Heymans*
*PReCISE Research Center,University of Namur, Belgium, Emails: thomas.loise@student.unamur.be,
xavier.devroey@unamur.be, gilles.perrouin@unamur.be, patrick.heymans@unamur.be
†SnT, SERVAL Team, University of Luxembourg, Email: michail.papadakis@uni.lu

***Abstract*—Mutation analysis forms a popular software analysis technique that has been demonstrated to be useful in supporting multiple software engineering activities. Yet, the use of mutation analysis in tackling security issues has received little attention. In view of this, we design security aware mutation operators to support mutation analysis. Using a known set of common security vulnerability patterns, we introduce 15 security-aware mutation operators for Java. We then implement them in the PIT mutation engine and evaluate them. Our preliminary results demonstrate that standard PIT operators are unlikely to introduce vulnerabilities similar to ours. We also show that our security-aware mutation operators are indeed applicable and prevalent on open source projects, providing evidence that mutation analysis can support security testing activities.**

***Keywords*-Mutation analysis; Mutation operators; Security Testing; PIT; FindBugs**

I. Introduction

Mutation testing is a popular fault-based testing technique [1], [2]. As every fault-based technique, it provides guarantees that the software under analysis is free from specific types of faults [3]. The technique has attracted a lot of interest because it forms a flexible and effective way to perform testing. Thus, it is used to guide test generation [4], to perform test assessment [5] and to uncover subtle faults [6]. It works by making syntactically altered program versions of the system under test. These alterations are designed to reflect the faults that our testing seeks for and are used for assessing the adequacy of testing. The approach is flexible because it relies on the introduced alterations [2]. Thus, by designing appropriate mutations it is possible to test all structures of a given language and almost everything that testing process seeks for. In view of this, we design security aware mutations that can be used to guide the testing of security related issues. Taking advantage of the fault-based nature of the technique, our mutations ensure that security-aware faults are not present and through regression tests that these will not appear in the future when the software will evolve. Existing mutation operators, especially those used by the Java mutation testing tools [7] are restricted to simple syntactic alterations and faults. Hence, it is unlikely that they can lead to tests that effectively exercise security related aspects of the applications. To deal with this issue, we design security-aware mutations based on common security bug patterns encoded by a well-known static analysis tool called FindBugs-sec-plugin[1] [8]. The bug patterns used by the static analysis tools aim at identifying potential issues with the code under analysis. Therefore, they point-out the presence of potential bugs and not opportunities for injecting them as it is done by the mutation operators. To cover this last point and design our security-aware mutations, we manually analyzed the bug patterns, inferred the classes of faults they represent and inverted them, *i.e.*, we defined rules that introduce these defects. We have implemented our mutations on PIT mutation testing engine [9] and provide initial exploratory results showing its applicability and difference from the traditional mutation operators. Thus, we applied both the traditional and our operators on four subject programs and validated the presence of potential vulnerabilities using FindBugs. Our results demonstrate that traditional operators are ineffective in introducing such security-aware faults.

[1]Find Security Bugs is a plugin for FindBugs and aims at identifying security issues in Java web applications.

Overall, our security related faults represent simple vulnerabilities, which can form an initial step for defining security-aware testing requirements. We believe that these requirements are particularly useful when building regression test suites of web application. Furthermore, our operators can be particularly useful in evaluating and comparing fuzzing or other security testing tools. In summary, our paper makes the following contributions:

1) We design 15 security-aware mutation operators for supporting security mutation testing.
2) We extend PIT so that it applies both traditional and security-aware mutation testing. To support future research, we make our implementation publicly available.
3) We make an initial assessment of our operators demonstrating their prevalence and potential weaknesses of the traditional operators using large real-world projects.

The rest of the paper is organized as follows: Sections II and III presents operator definition process and detail our security-aware operators. Section IV describes our assessment and results. Sections V and VI discus threats to validity and related work. Finally Section VII concludes the paper.

II. Mutation Operators Definition Procedure

Security related issues have received little attention by the mutation testing literature. As a result, it lacks operators that introduce security bugs. While security bugs are more or less well understood, there is no clear definition that we could use. Therefore, for the purposes of this study we use the following definition: a *security bug* is a piece of code that can lead to one or several vulnerabilities in an application.

TABLE I: Security-aware Mutation Operators

Acronym	*Name*
UPPRNG	USE_PREDICTABLE_PSEUDO_RAND_NUM_GEN
RPTS	REMOVE_PATH_TRAVERSAL_SANITIZATION
UWMD	USE_WEAK_MESSAGE_DIGEST
RHNV	REMOVE_HOST_NAME_VERIFICATION
XMLPVXXE	XML_PARSER_VULNERABLE_TO_XXE
XMLPVXEE	XML_PARSER_VULNERABLE_TO_XEE
REIS	REMOVE_ENCRYPTION_IN_SOCKET
UC	UNSECURE_COOKIE
RHTTPOFC	REMOVE_HTTPONLY_FROM_COOKIE
URSAWSK	USE_RSA_WITH_SHORT_KEY
UBFWSK	USE_BLOWFISH_WITH_SHORT_KEY
PSQLI	PERMIT_SQL_INJECTION
UDESISE	USE_DES_IN_SYMMETRIC_ENCRYPTION
UECBISE	USE_ECB_IN_SYMMETRIC_ENCRYPTION
RRS	REMOVE_REGEX_SANITIZATION

Research on software security developed a number of techniques to identify vulnerabilities in source code. One such (effective) technique is based on static analysis and seeks to identify occurrences of problematic code patterns. Such tools detect security bugs by highlighting potentially vulnerable code based on common vulnerability patterns. In view of this, we propose to leverage their knowledge and gather a set of common security bugs, which we can turn into injectable faults. These faults can form our mutants and support security testing.

By gathering the security patterns supported by known static analysis tools we can identify certain types of security related faults. Unfortunately, these patterns only detect the presence of a potentially vulnerable code and not the transformation needed to inject a vulnerability. Indeed, a security mutation operator can identify a non-vulnerable code pattern and turn it into a vulnerable one. We transformed every occurrence of the vulnerability patterns in its non-vulnerable functional equivalent one in order to define our mutation operators. This was not a trivial task as it required manual analysis and comprehension of the vulnerability classes.

For the purposes of the present study, we used the security patterns of a well-known static security bug analyzer, named *FindBugs-sec plugin*. All the patterns we used are described in the plugin documentation [8]. We believe that these patterns are suitable for our purposes as most of them form real-world security bugs that are well justified by Findbugs-sec, with CVE and NIST references. We detail our operators in the following Section.

III. Security-aware Mutation Operators

Table I presents the acronyms and a short description of our mutation operators. For each operator, we provide its application *context*, its *goal*, and some *implementation* details, *i.e.*, how it proceeds to introduce a vulnerability in the application under test.

Use predictable pseudo random number generator (UPPRNG). *Context:* Pseudo Random Number Generators (PRNGs) are widely used in secure-aware contexts and especially in cryptography to avoid prediction that could ease undesired decryption. *Goal:* the `UPPRNG` operator tries to make the application vulnerable to predictable random number generator attacks potentially leading to various security leaks (authentication, authorization, *etc.*). *Implementation:* this operator replaces the unpredictable pseudo random generators from the `SecureRandom` class by predictable ones using the `Random` class.

Remove path traversal sanitization (RPTS). *Context:* web applications often provide internal file access functionalities to their external users by requiring them to provide the desired file's name. *Goal:* the `RPTS` operator introduces a vulnerability which allows a malicious user to enter a path to access directories or files regardless of the file access policy defined by the web application. *Implementation:* the operator simply removes calls to input file names sanitization functions, generally used to avoid this vulnerability.

Use weak message digest (UWMD). *Context:* message digests, or hashing functions, are very often used to assure the integrity of received data. However, some hash functions are weak because of their high collision degree: in this case, for a hashed string, a malicious user can easily craft another string producing the same hash. *Goal:* the `UWMD` operator introduces a vulnerability in integrity checking of received data by using a weak hash function (*i.e.*, MD5). *Implementation:* it identifies hash function calls and replaces them by MD5 hashing.

Remove host name verification (RHNV). *Context:* a web application needing to authenticate its clients may verify their host names, usually after a successful SSL handshake. *Goal:* the `RHNV` operator removes this authentication, making the application vulnerable to man-in-the-middle attacks. *Implementation:* it removes standard methods used to authenticate clients using their host names.

Make XML parser vulnerable to XML Entity Expansion attack (XMLPVXEE). *Context:* web services often parse XML documents, to communicate with other web services in a standardized way. A known attack is the *billion laughs* attack which is an instance of a denial of service attack on XML parsers that require them to exponentially expand the tree with dummy text ("LOL"). However, one can prevent this attack by enabling a standard security option on the XML parser. *Goal:* the `XMLPVXEE` operator introduces a vulnerability in external XML parsers to expose the application to DOS attacks. *Implementation:* the operator disables standard security options of the XML parsers just before the XML parser begins parsing. It performs this task by identifying methods used on standard XML Java parsers, like `XMLReader` or `SAXParser` instances.

Make XML parser vulnerable to XML eXternal Entity attack (XMLPVXXE). *Context:* for this operator, in addition to the XML document parsing feature, we also assume that the attacker has a way to access the result of the parsing. In this context, an XML eXternal Entity (XXE) attack can lead to a confidentiality leak by accessing unauthorized files. To prevent this attack, developers have to enable a standard security option on their XML parsers. *Goal:* the `XMLPVXXE` operator introduces a vulnerability in external XML parsers to expose the application to XXE attacks. *Implementation:* it disables standard XXE security option of the XML parsers before an XML parsing. Like the `XMLPVXEE` operator, the `XMLPVXXE` operator performs this by identifying methods used on standard XML Java parsers.

Remove encryption in socket (REIS). *Context:* it is very usual in web applications to exchange encrypted data (*i.e*, passwords, e-mail addresses, *etc.*) with a user. This is commonly done by using sockets encrypting data with SSL on HTTP. *Goal:* the `REIS` operator tries to weaken the application's sent data to expose it to a confidentiality leak. *Implementation:* it removes the SSL encryption in sockets by identifying and removing standard Java SSL-encryption. For instance, it can replace sockets created with a `SSLSocketFactory` by sockets created with a `SocketFactory`.

Unsecure cookie (UC). *Context:* cookies are defined by the HTTP protocol as pieces of information sent by the server to the client's browser. Some cookies can store secret values proving the authentication of the client and must therefore be encrypted using SSL during communication. Cookies are meant to be sent by the browser with each request from the client, disregarding the secured-nature of the communication. To make sure that a browser will not make the mistake of sending a sensitive cookie in an unsecured HTTP communication, a *secure flag* can be set on the cookie, asking the browser to send this cookie only during HTTPS communications. *Goal:* the `UC` operator allows to send sensitive cookies during unsecured HTTP communication. *Implementation:* it removes the call to the methods setting the secure flag on cookies.

Remove HTTP-only flag from cookie (RHTTPOFC). *Context:* even if a cookie was sent using an HTTPS communication, web pages' scripts can access it on the client-side by asking the browser concrete access to the session's cookies. An attacker may access those cookies on the client-side by using a cross-site scripting (XSS) attack. To prevent this, cookies have an *HttpOnly flag* asking the client's browser to not share this cookie with scripts. Of course, the flag is just mitigating the risk, since it relies on the trust in the browser. *Goal:* The `RHTTPOFC` operator exposes the web pages to such session's cookies confidentiality leaks. *Implementation:* It removes the call of standard methods setting the httpOnly flag on cookies, allowing to share the cookie with client-side scripts.

Use RSA with short key (URSAWSK). *Context:* RSA is an asymmetric encryption algorithm used in web applications to exchange confidential data. Over time, with the improvement of computation power, the RSA algorithm needs longer keys to keep the exchange secured and to resist to brute force attacks. NIST[2] recommends the RSA keys to be at least 2048 bits long. *Goal:* the `URSAWSK` operator tries to weaken RSA encryption to make brute force attacks possible, allowing confidential data to leak. *Implementation:* it detects the use of RSA encryption with a sufficient key size and sets its to 512 bits.

[2]http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf

Use Blowfish with short key (UBFWSK). *Context:* Blowfish is a variable key size symmetric encryption algorithm five times faster than triple DES. Just like RSA, Blowfish could also be used with a short key (less than 128 bits). *Goal:* The `UBFWSK` operator tries to weaken Blowfish encryption to expose the application to brute force attacks, also allowing data to confidentiality leaks. *Implementation:* When the operator detects the usage of Blowfish with a sufficient key size, it sets the key size to 64 bits.

Permit SQL injection (PSQLI). *Context:* SQL injections exploit the fact that the web application uses input(s) from the user to build an SQL query that will be executed by a Data Base Management System (DBMS). The idea for an attacker is to inject SQL code in the input used to build the query in order to maliciously alter the database and or get access to private information. To prevent these attacks, Java APIs provide methods to prepare/encode queries and send them without their external inputs to the DBMS, requiring these inputs separately from the user. Inputs are then used to finalize construction of the query and execute it. Hence, it is not possible anymore for an attacker to play with the SQL-syntax to create tainted queries. *Goal:* the `PSQLI` operator tries to weaken the web application's protection against SQL-injection attacks to expose it to various security leaks potentially threatening its confidentiality, integrity, and authentication mechanisms. *Implementation:* it detects usages of SQL injection-proof APIs and replaces such usages by unsecured APIs in order to execute SQL queries.

Use DES in symmetric encryption (UDESISE). *Context:* in secured web applications, symmetric encryption is often very valuable to exchange sensitive data with external users. Data Encryption Standard (DES) was a popular symmetric encryption algorithm recognized as now sensitive to brute force attacks due to the great advances in computer performances. Therefore, web applications should prefer other symmetric encryption algorithms, like AES. *Goal:* the idea of the `UDESISE` operator is to weaken the confidentiality of symmetrically encrypted data, exposing it to leaks. *Implementation:* it detects the usage of a symmetric encryption algorithm and replaces it with DES encryption. This operator requires to modify several Java code lines. Though, PIT's architecture wasn't designed for this kind of modifications. Therefore, `UDESISE` is still under review but our initial implementation provides promising results.

Use ECB in symmetric encryption (UECBISE). *Context:* symmetric encryption may be done using different modes, describing how the algorithm should encrypt a message, which is split into blocks of fixed size. The Electronic CodeBook (`ECB`) mode encrypts two identical blocks into two identical ciphered blocks, introducing redundancy in the encrypted message, which makes it easier for an attacker to decrypt the message. *Goal:* the `UECBISE` operator tries to weaken the confidentiality of symmetrically encrypted data by easing its decryption using ECB mode. *Implementation:* it detects the usage of a symmetric encryption algorithm and replaces its mode by `ECB`.

Remove regex sanitization (RRS). *Context:* modern websites are following the idea of WEB 2.0, enabling the participation of external users to the content of a web page. Thus, web applications can have a lot of stored data coming from external users. To prevent malicious users' content, web applications commonly validate the inputs coming from external sources using regular expressions. *Goal:* the `RRS` operator tries to introduce vulnerabilities in external input filters of a web application. *Implementation:* it detects regular expressions usages and replace them by a dummy expression, which is always true.

IV. Evaluation

In this section, we report on our preliminary efforts to assess the relevance our set of security-aware mutation operators. To this end, we state three research questions: *(RQ1)* Are the standard operators of PIT likely to introduce vulnerabilities? *(RQ2)* Does our mutation operators introduce vulnerabilities that are detectable by the FindBugs' static analysis? *(RQ3)* How prevalent is the application of our mutation operators on open source projects?

Case Studies

In order to answer the different research questions, we considered the following case studies:

1) iTrust: iTrust[3] is a web application developed and maintained by the students of NCState University and consists of 24,785 lines of code. It provides a platform accessible to patients and doctors, to keep track of the patient's medical history.

2) Vuze-Azureus: Vuze[4] is a popular open-source Bittorrent client, consisting of 186,247 lines of code.

3) OpenLegislation: OpenLegislation[5] is an open source web application developed and maintained by the New York State Senate. The goal of this application is to give access to several NYS's data including bills, resolutions and laws. It consists of 912 classes and is 33,819 lines of code.

4) AntsP2P: Ant's Peer-to-Peer is an open-source Bittorrent client[6] (consisting of 19,399 lines of code), like Vuze.

RQ1: PIT Operators and Vulnerabilities

To investigate this question, we generate mutants using PIT's standard mutation operators and use FindBugs to count the vulnerabilities found. This metric gives an indication of the suitability of such operators to cover vulnerabilities even if they are not designed for that. We selected the iTrust case for this assessment: we elicited a sample of 33 classes, for which our security-aware operators yielded vulnerabilities found by FindBugs, and applied PIT on those classes. Table II records our results.

Overall, PIT generated 5486 mutants and introduced only two vulnerabilities (see Table II). The introduced vulnerabilities relate to potential SQL injection attacks that can occur by inserting dynamically generated strings in a query (`SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE`) or by removing a conditional execution (`SQL_INJECTION_JDBC`). Though these are genuine security issues, this harvest with standard operators appears to be mediocre. As we will see in the following sections, our operators are able to introduce a much larger number of vulnerabilities (62 for iTrust).

[3]https://sourceforge.net/projects/itrust/ (version 21.0.01)
[4]https://sourceforge.net/projects/azureus/ (version 5.7.40)
[5]https://github.com/nysenate/OpenLegislation (version 2.2)
[6]https://sourceforge.net/projects/antsp2p/ (version beta1.6.0)

TABLE II: Mutating iTrust with PIT's standart operator set. The table records the number of mutants and vulnerabilities that were generated per used operator.

Operator name	*#mutants*	*#Vulnerabilities*
ArgumentPropagationMutator	42	0
ConditionalsBoundaryMutator	48	0
ConstructorCallMutator	431	0
IncrementsMutator	12	0
InlineConstantMutator	696	0
MathMutator	41	0
MemberVariableMutator	83	0
NegateConditionalsMutator	368	0
NonVoidMethodCallMutator	1539	1
RemoveConditionalMutator_EQUAL_ELSE	320	0
RemoveConditionalMutator_EQUAL_IF	320	0
RemoveConditionalMutator_ORDER_ELSE	48	0
RemoveConditionalMutator_ORDER_IF	48	1
RemoveIncrementsMutator	12	0
RemoveSwitchMutator	15	0
ReturnValsMutator	289	0
SwitchMutator	2	0
VoidMethodCallMutator	1172	0
Total	5486	2

RQ2: Static detection of Vulnerabilities

Our second research question investigates the extend to which our security-aware operators introduce vulnerabilities and if they are always detectable statically with FindBugs. This process can be seen as a sanity check of the function of the implemented operators. To perform this check, we manually created a sample project containing on average one class that contains one application instance of the mutation operators we implemented. Each class was implemented so that it can trigger one specific mutation. We used PIT's mutation engine to generate the mutants with respect to our operators. We wrote scripts to keep track of the applied mutations and results obtained via FindBugs. We specifically tuned FindBugs to check every potential issue, even those of low confidence, at the cost of performance.

The analysis results are reported in Table III. `RRS` and `UDESISE` were not evaluated. `RRS` was not inspired by a FindBugs pattern. Regarding `UDESISE`, it requires higher order mutation and its correct operation is currently experimental in PIT. Overall, we can see that 9/13 (nearly 70%) operators generated a vulnerability that could be found by FindBugs. For the four remaining ones, non-recognition causes are:

a) Remove Host Name Verification (RHNV): the vulnerability is actually introduced, because the replacement of `HostNameVerifier.verify(...)`'s result by `true` was present in the mutant. It is not recognized because FindBugs identifies a vulnerable HostNameVerifier by

TABLE III: Injected classes of vulnerabilities that were identified by FindBugs (with the security plugin), in a sample project. We mutated this project and verified the presence of vulnerabilities by comparing the static analysis reports of the mutants and the original programs. Y signifies that the injected vulnerability was identified by FindBugs.

	UPPRNG	*RPTS*	*UWMD*	*RHNV*	*XMLPVXXE*
Recognized?	Y	Y	Y	N	N
	XMLPVXEE	*REIS*	*UC*	*RHTTPOFC*	*URSAWSK*
Recognized?	N	N	Y	Y	Y
	UBFWSK	*PSQLI*	*UDESISE*	*UECBISE*	*RRS*
Recognized?	Y	Y	/	Y	/

looking at its code. Indeed, for FindBugs, a vulnerable `HostNameVerifier` is identified by a constant `return true;` in all its `.verify(...)` methods. Because we didn't mutate the `HostNameVerifier.verify(...)` method but its calls, FindBugs could thus not recognize it.

b) Xml Parser Vunerable to XXE/XEE (XMLPVXXE/ XMLPVXEE): the vulnerability is also present as we remove a secure parsing feature of the XML reader. However, we mutated a secure original program that had the following line: `XMLReader.setFeature(SecurizingFeature, true)` prior to parsing. Therefore, the mutant had the following code:

```
XMLReader.setFeature(SecurizingFeature,
    true);
XMLReader.setFeature(SecurizingFeature,
    false);
XMLReader.parse(input);
```

We hypothesize that when FindBugs analysed the first line, it directly returned that the vulnerability was eliminated.

c) Remove Encryption In Socket (REIS): here, after performing several tests, we supposed that FindBugs identifies unsecured sockets only by the following pattern:

```
Socket s = new Socket(address,port);
```

However, since we created a unsecured socket in the mutant with the following method:

```
Socket s = SocketFactory.getDefault().
    createSocket(address,port);
```

and upon manual inspection of `createSocket(...)` method's specification, we certify that the two manners of creating an unsecured socket are equivalent.

Regarding the results of this section, we can answer RQ2 by stating that all vulnerabilities were actually introduced though only 70% were found by FindBugs. Missed vulnerabilities either stem from the static analyzer's incompleteness `(REIS)` or optimisations `(XMLPVXXE/XMLPVXEE)`. Regarding `RHNV`, this vulnerability requires a global (e.g., analysing the call graph) or a dynamic reasoning, techniques that are out of reach for FindBugs. Although we took inspiration on FindBugs patterns to create our mutation operators, they can trick FindBugs. Therefore, our operators might be used to validate static analysis tools as well.

TABLE IV: Mutating projects with new operators

	iTrust	*VUZE*	*OPENLEGISLATION*	*ANTSP2P*
#classes in input	405	4669	912	406
#classes mutated	33	30	57	9
#mutants generated	62	57	154	18

TABLE V: Number of security-aware mutants generated on 4 open source projects. The table entries record the number of mutants generated per mutation operator and project. Non referenced operators did not produce any mutants.

	UPPRNG	*UWMD*	*PSQLI*	*UECBISE*	*RRS*	*TOTAL*
iTrust	1	0	39	0	22	62
VUZE	8	2	0	9	38	57
OPENLEGISLATION	0	0	0	0	154	154
ANTSP2P	2	4	0	11	1	18

RQ3: Prevalence of Security-aware Operators' application

The preceding research questions were meant to assess the relevance of our security-aware operators. However, the question that it is raised here is how numerous these vulnerabilities are. In practice, it is hard to trigger and secure vulnerabilities and thus, it is possible that injecting a large number of them may be excessively expensive. To assess this point, we simply generated mutants with our operators for the four projects we considered. Table IV records the results provided by our 15 operators.

A first observation is that five of our operators are prevalent in practice. Interestingly, these operators are not numerous indicating that mutation-based security testing is feasible. Ten operators were not applicable in the selected projects.

Another interesting point is that the injected faults only concern a fraction of the project's classes (e.g., 0.67% for VUZE). Table V shows the repartition of applied operators for each project. There are disparities, `PSQLI` that accounts for 61% of the mutants in iTrust and does not appear in other projects, while `RRS` appears in all projects. `RRS` "popularity" is also quite understandable as a regex-sanitization function can be assumed to be more common to web applications than the use of `Blowfish`, for instance. We therefore conclude that many security-aware faults appear in the selected projects. Though, we note that further case studies are required to gain confidence on the prevalence of our operators.

V. Threats to validity

Internal validity. One possible threat of our study is due to the used tools. We implemented our operators in PIT, which relies on Java bytecode manipulation (using ASM Java bytecode framework as an abstraction layer) to perform mutations. Thus, potential defects may influence our results. To verify that our operators were correctly implemented, we used code review on the operators' implementation, and, for each experiment, we manually inspected the generated mutants. Moreover, we checked in *RQ2* that the vulnerabilities from the generated mutants are detected by FindBugs: it is the case for nine of them, explanations for the four remaining ones are given in Section IV.

Construct validity. We chose to use iTrust to answer *RQ1*, because of the high number of security concerns that the application has to take into account. This tends to be confirmed by the higher percentage (8%) of classes mutated by our new operators in Table IV. *RQ2*'s only goal was to validate our operators' implementation. Therefore, we used ad hoc classes, one per operator, which is enough for this purpose. For *RQ3*, we took 4 open source projects: 2 web applications and 2 Bittorrent clients. We plan to extend the number as well as the diversity of the considered projects in our future work.

External validity. We designed our mutation operators, based on patterns defined in FindBugs sec plugin. Therefore, it is questionable whether these are usually met in practice. However, each of these patterns introduces one or more vulnerabilities described in well-known vulnerability reporting authorities (like NIST and CVE), as referenced in FindBugs documentation[7]. To perform our evaluation, we selected 4 open source projects in which security is a key concern.

VI. Related Work

Using mutation for security purposes was explored at the model-level by Mouehli *et al.* [10] where the authors mutate access control models to qualify security test suites. Operators change user roles and allowed actions, deleting policy rules or modify their application context. Dadeau *et al.* defined operators that introduce leaks in a high-level security procotol [11]. Büchler *et al.* considered mutating the abstract model of a web application by removing authorization checks and un-sanitizing data [12], but they do not detail the operators. Although, these operators are inspired from actual vulnerabilities, as being model-based they model different defects from our code-based ones.

To the best of our knowledge, there is no set of security-aware mutation operators for Java. Perhaps the closest related work is that of Nanavati *et al.* [13], Shahriar and Zulkernine [14] and Ghosh *et al.* [15] that defined mutation operators related to the memory related faults. All these operators introduces memory manipulation issues in C programs (such buffer overflows, uninitialized memory allocations and etc.), which may be exploited by security attacks. As these operators make heavy use of memory allocation primitives, specific to the C language, they are rather different from ours.

VII. Conclusion

This paper introduces security-aware mutation testing operators. Inspired by common vulnerability patterns, we have designed 15 new mutation operators for Java, which we implement in the mutation testing engine of PIT. We used FindBugs and its security plugin to assess whether standard PIT mutation operators are likely to introduce vulnerabilities, like those supported by our operators, and demonstrated that they fail to do so. Our case studies validated the purposes of our operators and revealed that certain types of vulnerabilities are prevalent in open source projects.

[7]http://find-sec-bugs.github.io/bugs.htm

This work constitutes the first step towards a relatively new direction of mutation testing research which is the *mutation-based security testing*. With the use of our mutants, security related test suites can be designed and documented. Other potential uses of our mutation operators are in education and in the systematic evaluation and comparison of fuzzing and other security testing tools. Overall, our goal is to use mutation to define adequacy criteria for security testing.

In the future, we plan to extend our work towards the following directions: First, we plan to consider a much larger set of security patterns that we will mine from open source projects. Second, we will assess our research questions on more subjects and confirm our observations with actual tests from fuzzing tools. Third, we would like thoroughly assess the practical benefits of our security testing metrics.

References

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[2] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[3] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.

[4] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, 2010, pp. 121–130.

[5] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 354–365.

[6] T. C. Thierry, M. Papadakis, Y. L. Traon, and M. Harman, "Empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *ICSE*, 2017.

[7] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, "Analysing and comparing the effectiveness of mutation testing tools: A manual study," in *International Working Conference on Source Code Analysis and Manipulation*, 2016.

[8] P. Arteau, "Bug patterns - find security bugs http://find-sec-bugs.github.io/bugs.htm."

[9] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 449–452.

[10] T. Mouelhi, Y. L. Traon, and B. Baudry, "Mutation analysis for security tests qualification," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, Sept 2007, pp. 233–242.

[11] F. Dadeau, P.-C. Héam, R. Kheddam, G. Maatoug, and M. Rusinowitch, "Model-based mutation testing from security protocols in hlpsl," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 684–711, 2015.

[12] M. Büchler, J. Oudinet, and A. Pretschner, "Semi-automatic security testing of web applications from a secure model," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, June 2012, pp. 253–262.

[13] J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke, "Mutation testing of memory-related operators," in *Software testing, verification and validation workshops (ICSTW), 2015 IEEE eighth international conference on*. IEEE, 2015, pp. 1–10.

[14] H. Shahriar and M. Zulkernine, "Mutation-based testing of buffer overflow vulnerabilities," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland*, 2008, pp. 979–984.

[15] A. K. Ghosh, T. O'Connor, and G. McGraw, "An automated approach for identifying potential vulnerabilities in software," in *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, 1998, pp. 104–114.