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.

https://doi.org/10.1109/ICSTW.2017.24

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.

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.

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

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.