Testing Security Properties in Java *

Padmanabhan Krishnan
Oracle Labs
Brisbane, Australia
paddy.krishnan@oracle.com

Jerome Loh
ITEE
The University of Queensland, Australia

Larissa Meinicke
ITEE
The University of Queensland, Australia
l.meinicke@uq.edu.au

ABSTRACT

In this paper we describe our initial experience of using mutation testing of Java programs to evaluate the quality of test suites from a security viewpoint. Our focus is on measuring the quality of the test suite associated with the Java Development Kit (JDK) because it provides the core security properties for all applications. We define security-specific mutation operators and determine their usefulness by executing some of the test suites that are publicly available. We summarise our findings and also outline some of the key challenges that remain before mutation testing can be used in practice.

1. INTRODUCTION

The Java security model provides language-level access control to security-sensitive resources and actions. Proper use of this model is the responsibility of the programmer, and errors may arise in its use. As there is no formal model of the desired security properties, techniques such as verification are not directly applicable. So testing is one way to detect such errors. However, we cannot be sure whether the tests themselves are thorough enough to catch all errors in the use of the security model. So the aim is to determine the quality of test suites that check for the security properties.

Mutation testing [8] can be used to measure the effectiveness of a test suite. The general idea is to seed faults into the original program to derive mutations and check whether the test suite can distinguish the behaviour of each of the mutations against the original program. The percentage of mutations that a test suite can distinguish, called the mutation-kill ratio, can be used as a measure of effectiveness of the test suite. As a result, a tester can determine the causes for not distinguishing a mutation to improve the test suite.

King and Offutt [9] were the first to define a formal set of 22 mutation operators for Fortran. The mutation operators included arithmetic operator replacement (e.g. '+' to '-'), logical connector replacement (e.g. AND to OR), variable replacement, and statement deletion. However, such operators are not adequate for object-oriented languages like Java. Mutation operators specific for object-oriented languages, such as deleting an overriding method, have been proposed [1, 3].

As the number of potential faults in a program can be very large, generating mutations that represent all the faults is not useful in practice. In order to reduce the number of generated mutations, one usually assumes a competent programmer hypothesis where the possible mutations are derived from a set of typical errors a competent programmer can make. For instance, not all variable replacement operators may be relevant in a given situation. Otherwise, mutations will, typically, not represent real faults [6].

In this paper, we outline the issues related to mutation testing in the evaluation of test suites that are relevant for Java security. We define mutation operators that are specific to Java security and present some preliminary results. We then highlight the challenges that remain before mutation testing can be used effectively by practitioners.

In Section 2 we summarise the Java security model, and in Section 3 we present our approach, including the operators and processes to test the Java Development Kit (JDK). Finally, in Section 4 we summarise the lessons learned and describe the limitations that need to be overcome.

2. BACKGROUND: JAVA SECURITY

The Java security model is one of access control—certain actions are designated as security-sensitive, requiring some set of permissions. Execution of these actions completes successfully only if the calling code possesses the required permissions. The set of required permissions is specified programmatically by the developer using permission checks. Errors could arise as the developer could omit or add a permission check, check the wrong permission etc. The set of permissions a class is granted is specified by a security policy associated with the program. Such a security policy allows the developer (or user) to grant permissions to classes depending on where they are loaded from (i.e., its codebase) [5]. The desired security policy is enforced by a security manager. When the program executes the method checkPermission(Permission perm), all classes on the call stack must have the permission. If this is the case, the call returns normally, otherwise a security exception is thrown.

Java also allows a class to grant some of the permissions it holds to its caller. The doPrivileged block demands that only the immediate caller of the the block has the requisite privileges—thus the classes below the immediate caller on the call stack are not checked. It is possible to pass an access control context (ACC) to a doPrivileged block where the permissions given to the block are derived by intersecting the permissions present in the various protection domains in the ACC. These protection domains represent the permissions associated with the classes of a potential call stack.

*Java and JDK are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

The ability to pass an ACC to a doPrivileged call allows a class to save a context and reuse it when executing some code. Such reuses are typically useful when handling callbacks where the context of callback execution must match the context of callback creation.

3. MUTATION TESTING OF THE JDK

The principal aim of this project is to develop a system to measure the quality of test suites for the JDK. The JDK is a library that provides the key security mechanisms that numerous applications rely on. Any security defect in the JDK has the potential to affect all these applications. We also intend to use the results of mutation testing to provide guidance to the test developer to develop new tests and deprecate unnecessary tests. So we focus only on mutating the code in the JDK, and do not consider mutating policy files that are part of the applications.

As the JDK is a library, the test suite has programs that invoke various API JDK calls. For security testing, the test suite will also set up the policy files granting permissions to the codebase. Note that we mutate only the JDK; not the code associated with the tests.

In a security context, mutation testing has been applied to detect string formatting and buffer overflow vulnerabilities in C programs [14, 13]. But these mutation operators are not related to any security model. Martin and Xie [12] as well as Traon et al. [15] have also extended mutation testing to testing access control policies. Both studies utilise a system based on rules that either permit or deny sensitive actions based on the caller's identity, or role. The mutation operators add or remove rules from a policy, and change the action and targets associated with a rule. Conceptually, these mutation operators are similar to what is desired for Java. However, we need operators that directly manipulate the Java constructs that deal with security.

Semantically, the mutation operators we consider can be classified as either narrowing or widening the set of permissions. The narrowing operators either check for more permissions or remove permissions that are available to the code. One can check for more permissions by replacing the permission in a checkPermission call (say p) with q such that q implies p. One can remove permissions by adding a more restrictive frame to the ACC given to a doPrivileged call. The widening operators either check for fewer permissions or grant more permissions to the code.

Replacing a permission p in a checkPermission call with a permission q such that p implies q results in widening, as does removing a restrictive frame from the ACC given to a doPrivileged call. Note that it is possible to have mutation operators that widen a narrowing (or narrow a widening) but they are not considered in our preliminary study.

We now describe the specific mutation operators that we have implemented towards the evaluation of some of the test suites from the OpenJDK distribution.

The mutations of a checkPermission call fall into one of the following categories.

- Delete the checkPermission call. This operation is a form of widening because it is as if the application has been granted the elided permission.

- Replace the argument of a checkPermission with another permission. Technically, this operator can generate many mutants, depending on the class of permissions considered. To simplify the initial experimentation, we replace the argument with only AllPermission. Because we are checking for more permissions than originally present, this mutation represents permission narrowing.

The mutations of a doPrivileged call fall into one of the following categories.

- If the original doPrivileged call has only one parameter (i.e., equivalent to a call with a **null** ACC), it has all the privileges of the immediate caller. So in this case, only permission narrowing is possible. The narrowing can be achieved by either converting the call to a two parameter version where an explicit ACC is provided, or to a var-args version where an explicit ACC along with an array of permissions is provided. As there are a large number of possible ACCs we consider only an ACC which has no permission. This transformation effectively treats the code in the doPrivileged block as unprivileged, and thus any checkPermission performed will fail. We consider only one explicit permission for the var-args version.

- If the original doPrivileged call has two parameters, i.e., the privileged action and an ACC, one can mutate it in the following ways:

  - Convert it to a var-args version with an explicit list of permissions that is demanded. For example, the call doPrivileged(act, acc) is converted to the var-args call doPrivileged(act, acc, p) where the specific permission p is demanded. Concretely, we use only the permission PropertyPermission("$*$", "read") for p. This conversion represents permission narrowing.
  - Replace the given ACC with another independently created ACC. Again, for tractability we consider only an ACC which has no permission as the replacement. This replacement is also a permission narrowing.
  - The third option is to replace the given ACC with an ACC that is derived from the provided one by eliding one of the protection domains. More specifically, we drop only the first protection domain although in general, we could drop any. Dropping a protection domain from the list of protection domains in the ACC can potentially widen the permissions available to the code. We call this mutation operator truncating the ACC.

3.1 Preliminary Results

We summarise the results of the experiments run on selected classes of the OpenJDK8-u66. As none of the existing tools were suitable for our experimentation, we adopted a manual process. Although the manual process was time consuming, the aim was to determine whether the mutation operators identified limitations in existing test suites. For each relevant class, we edit the source and replace each occurrence of the relevant construct with its mutation, and compile the modified source. The original rt.jar is then updated by replacing the original class with the compiled mutated class. The tests for the modified class are then executed using the jtreg infrastructure. If jtreg reports a difference between the number of tests that passed in the original and in the mutant, the mutant is considered killed; otherwise, it is considered to have lived.

We chose the following for the experimentation:

- java.security, which is the core of the access control model,

- javax.security.auth, which provides an authentication framework,

- Classes from java.lang, java.io and java.net, which make use of security-sensitive operations.

- Randomly chosen classes from java.util, java.nio.file, java.nio.charset.spi, java.nio.channels, and javax.swing.

An important limitation of these results that must be stressed is that the tests used are only those that are publicly available in the OpenJDK. Hence, the actual mutation-kill ratio is not directly useful; rather, patterns of mutations that have not been killed are useful to identify the type of tests that are essential to improve the quality of the testing process.

Table 1 presents the number of mutations generated by our operators and the number of mutations killed by the test suites.

Mutation Operator	Total Mutants	Mutations Killed
Deleting checkPermission	15	6
Checking AllPermission	59	21
No-permission ACC	20	8
Use of PropertyPermission	19	4
Truncating the ACC	15	0
Total	128	39

Table 1: Results

Table 2 presents the mutation-kill results for each test suite. The two security specific classes of javax.security.auth and java.security kill around 38% of mutations which is similar to the java.io package which also has many security-sensitive operations. Similarly, the tests associated with java.lang.Class kill 50% of the, admittedly, small number of mutations. But other classes, such as from the java.net package, kill just one mutant each. The low mutation-kill ratio is because the OpenJDK test suite does not focus on security.

Test Suite	Relevant Mutants	Mutations Killed
javax.security.auth	41	13
java.io	30	11
java.security	19	10
java.net	13	1
java.lang.Class	6	3
All others	14	1
Total	123	39

Table 2: Test outcomes from running mutants over relevant OpenJDK test suites

We summarise the key reasons that some of the mutants are not killed by the test suites. For this we do not consider equivalent mutants. Only five mutants (out of the 128 generated) were equivalent to the original program. All of them are from doPrivileged invocations. Two of the mutations are equivalent because the doPrivileged invocation occurs in a static initialisation block that is already privileged. Hence no difference in behaviour can be detected. The other three mutants are equivalent because the privileged action requires only one permission, which happens to be one that was granted.

The first predominant reason is that the mutant is not executed. That is, the test suites did not have the code coverage necessary to execute the mutation. The second reason is that some of the tests do not install a security manager. In such cases, removing the checkPermission calls has no effect. The third reason is that the tests considered do not consider the various combinations over permissions, and thus are unable to detect the checking of the wrong permission or missing permission checks. Some tests that detect security violations are reported as having failed as the output generated by the program is not different. For instance, security exceptions are caught but not reported. So the jtreg harness cannot report on the mutation being killed. Table 3 summarises the number of live mutants for each of the above reasons where CP-mutants and DoP-mutants refer to the mutants generated by altering checkPermission and doPrivileged calls respectively.

Reason	Number of Live	
	CP-mutants	DoP-mutants
Mutation code not executed	29	13
Lack of security manager	7	5
Insufficient permission coverage	18	12

Table 3: Reasons for live mutants

4. OPEN ISSUES

The 128 mutants (123 non-equivalent ones) that we have generated has shown the potential merit of the idea. However, a number of challenges remain. We outline some of the issues that might limit the adoption of mutation testing for this type of security testing.

4.1 Replacement Options

The first issue is related to permissions and ACCs. In our experimentation we have chosen a small set of constant permissions, e.g., AllPermission or PropertyPermission("∗", "read"). We have not considered permissions of different types, such as FilePermission or SocketPermission. Because the parameters to a permission object can be created with any possible string, the set of possible mutation options is prohibitive. One needs an algorithm to identify a relevant set of interesting permissions. For example, the set of interesting permissions can be from the set of permissions that are checked in the entire JDK. Thus no new permissions are introduced; however, considering only permissions in the JDK has the limitation that a defect caused by a permission that is relevant but not used by the developer will not be detected.

Similarly, there are numerous ACCs that can replace a given ACC. Here we have considered only a ACC with no permissions or an ACC derived by removing the first protection domain. In general it is possible to use a variety of values to create the protection domains. So there are any number of options for both the parameters to a ProtectionDomain and the number of elements in the ACC.

In general, because the number of potential replacements for a given mutation operator is very large, exploring all of them is impractical. A more precise formulation of the competent programmer hypothesis will help in identifying the set of relevant permissions or ACCs.

4.2 Equivalence and Minimal Mutations

Although our initial choice of mutation operators does not result in many equivalent mutants, extending the mutation operators might influence the number of equivalent mutants.

Even if the mutants are not equivalent, the number of test cases required to distinguish the various mutants might not be practical.

For example, replacing permission p with q such that either p implies q or vice-versa will demand that the test suite must detect "equivalence" of permissions. Such demands might be too strong. So it is better to handle the issue in the mutation generation process.

As noted in our results, tests that detect the security violation but do not report it result in mutations not being killed. One option is to install a custom security manager that logs all the method calls. If the logs are different, the test outcomes will be different and the mutation will be marked as killed.

Other related issues include finding the minimal set of distinct mutations [2] and thus eliminating mutations which are themselves equivalent or those that are subsumed by other mutations. As they note, the problem of reducing the set of mutations is NP-complete and thus approximations of the minimal set will be required.

4.3 Other Mutation Operators

One needs to explore other mutation operators. For example, one can either introduce or elide the ACC associated with a specific checkPermission call. That is, sm.checkPermission(p, acc) can be replaced by the call sm.checkPermission(p) which in turn can be replaced by sm.checkPermission(p, acc) where the value acc is some artificially created ACC.

4.4 Automation and Tool Support

While our manual process is sufficient for exploratory work, tools that can handle many mutants as well those that integrate with the testing scripts are necessary. Our manual approach modifies the source code, which requires one to compile each class separately and replace the class in rt.jar to run the test. This is very time consuming as the compilation and class replacement has to occur for each mutant. A semantically equivalent but more efficient technique is required, especially if the approach has to scale to the entire JDK. For example, a tool that can edit the byte code (for instance, similar to BCEL described in the book by Horstmann and Cornell [7]) might be useful. A mechanism to have different versions of the same class within one test cycle would also be useful.

Although there are tools such as *muJava*[1], [11] Jumble[2] and PIT[3], none of them were suitable for security testing. For instance, *muJava*'s method-level operators allow deletion of statements, but changes to the tool are required to specify deletion of only the checkPermission calls. Otherwise, the tool generates too many non-security specific mutations. A comparison of various tools [4] focuses on general mutation testing and is not directly applicable for security specific testing. Similarly, [10] show that PIT does not capture many of the mutants created by experts. The ability to configure existing mutation tools to be able to select the mutation operators as well as the locations where they need to be applied is essential. In other words, the ability to specify the competent programmer hypothesis will enable the user to generate only relevant mutations.

5. CONCLUSION AND FUTURE WORK

In this paper we have identified a set of security-focused mutation operators. Initial investigation indicates that these targeted operators produce few equivalent mutants and are useful for measuring the quality of a test suite.

The ultimate measure is the ability to run all the tests (open and otherwise) on the relevant mutations generated from all the classes in the JDK. Such an experiment will provide a more useful insight into the strengths and limitations of using mutation testing as a quality measure and also help identify the improvements necessary for security testing.

[1] https://cs.gmu.edu/ offutt/mujava/

[2] http://jumble.sourceforge.net/

[3] http://pitest.org/

6. REFERENCES

[1] R.T. Alexander, J.M. Bieman, S. Ghosh, and B. Ji. Mutation of Java objects. In *Proceedings of the 13th International Symposium on Software Reliability Engineering, 2002 (ISSRE '02)*, pages 341–351, 2002.

[2] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 21–30, 2014.

[3] P. Chevalley and P. Thévenod-Fosse. A mutation analysis tool for Java programs. *International Journal on Software Tools for Technology Transfer*, 5(1):90–103, 2003.

[4] M. Delahaye and L. du Bousquet. A comparison of mutation analysis tools for Java. In *13th International Conference on Quality Software, 2013 (QSIC '13)*, pages 187–195, 2013.

[5] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security*. The Java Series. Addison Wesley, 2003.

[6] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering, 2014 (ISSRE'14)*, pages 189–200, 2014.

[7] C. S. Horstmann and G. Cornell. *Core Java, Volume II–Advanced Features*. Prentice Hall, 9th edition, 2013.

[8] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[9] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software – Practice & Experience*, 21(7):685–718, 1991.

[10] T. Laurent, A. Ventresque, M. Papadakis, C. Henard, and Y. L. Traon. Assessing and improving the mutation testing practice of PIT. *CoRR*, abs/1601.02351, 2016.

[11] Y-S. Ma, J. Offutt, and Y-R. Kwon. MuJava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[12] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 667–676, 2007.

[13] H. Shahriar and M. Zulkernine. Mutation-based testing of buffer overflow vulnerabilities. In *Computer Software and Applications, 2008 (COMPSAC '08)*, pages 979–984, 2008.

[14] H. Shahriar and M. Zulkernine. Mutation-based testing of format string bugs. In *High Assurance Systems Engineering Symposium, 2008 (HASE '08)*, pages 229–238, 2008.

[15] Y. L. Traon, T. Mouelhi, and B. Baudry. Testing security policies: going beyond functional testing. In *Proceedings of the 18th IEEE International Symposium on Software Reliability, 2007 (ISSRE'07).*, pages 93–102, 2007.