Proceedings of The National Conference
On Undergraduate Research (NCUR) 2019
Kennesaw State University
Kennesaw, Georgia
April 11-13, 2019

Metamorphic Testing for Software Security: Testing and Validating TripleDES

Nate Tranel
Computer Science
Montana State University Bozeman
Culbertson Hall, 100
Bozeman, Montana 59717 USA

Faculty Advisor: Dr. Upulee Kanewala

Abstract

The objective of this study was to learn more about the Metamorphic Testing (MT) technique. In MT, specific properties of a program under test are identified and labeled metamorphic relations (MRs). A source test suite is built, and the MRs are used to build a follow-up test suite. This entire test suite should pass on the original source code. The source code is then changed, or mutated, using specified mutation operations that are appropriate to the language. Ideally, the test suite should then fail on the mutated code, "killing" the mutants. When mutants survive this process, it helps the developer to identify flaws in the test suite. This study was conducted on an open-source implementation of TripleDES in ANSI-C. Using an open-source mutation engine called MUSIC, 6227 mutants were generated from the source code. A test suite was built using 3 identified MRs and 32 test cases initially. Three rounds of tests were executed: the first round established a baseline and determined which mutants did not compile correctly or had runtime errors, which were thrown out; during the second round, a fourth MR was introduced to the test suite; and during the final round, the test suite coverage was expanded. After the first round of tests and the filtering process, about 40% of the mutants were killed. After improving the test suite with a fourth MR, the score increased to about 47%. After improving test coverage, the final score was about 76%. All told, this approximately matched the performance of other MT studies. This study further demonstrated that MT is effective, though it is important to couple it with good test coverage. Assessing the mutants and comparing those killed versus those that survived gave some insight into an effective test suite for TripleDES.

Keywords: Metamorphic Testing, TripleDES, Software Security

1. Introduction

With the Internet playing a more crucial role than ever in people's lives, it is important that the things people do on the Internet are kept secure. Most software currently uses encryption algorithms in some capacity to achieve security. These algorithms can be difficult to test, however, since they may not have an output that is explicitly known to be correct. One solution to this is using Metamorphic Testing (MT), a testing approach that attempts to verify test results via relationships between the program outputs rather than verifying their individual values[2]. To verify that MT was an appropriate technique for cryptographic algorithms, the TripleDES algorithm was tested, as it has been widely used and well documented[12].

2. Background

Some information about the concepts used in this paper will be discussed here. If already familiar with TripleDES and Metamorphic Testing (MT), Section 3 contains the project structure.

2.1 TripleDES

TripleDES is a symmetric-key block cipher derived from DES (Data Encryption Standard). It was first published in 1998 ANSI ANS X9.52[10]. TripleDES essentially runs through DES three times, decreasing the chance that a cipher could be cracked through brute force. TripleDES uses three 64-bit keys with 8 bits of parity K_1, K_2, K_3, to create a ciphertext defined in equation (1) below.

$$C = Encrypt_{K3}(Decrypt_{K2}(Encrypt_{K1}(plaintext))) \tag{1}$$

Decryption is the opposite process; that is, each encrypt function is a decrypt function and vice versa, and the keys are used in the same order. As may be expected, there are a few ways to configure the keys: they can all be independent, which is the most secure method; two keys can be the same, which generally is not used frequently; and all can be the same, which is the least secure method[7]. With all independent keys, TripleDES has an effective key length of 112 bits. Since modern hardware can search this space in a reasonable amount of time, this algorithm is now considered broken, and has been effectively replaced by AES (Advanced Encryption Standard)[5].

2.2 Metamorphic Testing

Metamorphic Testing (MT) is a newer property-based software testing technique invented by T.Y. Chen et al in a 1998 paper and continually revisited[2]. It was designed with the idea of addressing the "Test Oracle" problem, which states that generally tests require some sort of "oracle" to assess correctness of testing outputs in black-box testing. Usually the oracle is a human, making the process difficult to automate. In MT, Metamorphic Relationships (MRs) of the Program Under Test (PUT) are identified to serve as a functional oracle. This means that certain properties unique to the PUT and identified by the human can serve as an oracle once implemented. This does require some setup, however. Though it is difficult to automate the subjective judgement of a human oracle, the MRs define rules that a human oracle would use.

MRs are derived from specific properties of the PUT. For example, if a program kept track of the total number of users visiting a website on a specific day, then the input to the program would be that day and the output would be the total number of visitors from that day. If the criteria changed to display the number of visitors in the past hour, that number and those specific visitors would have to be a subset of the total, so the oracle could safely reason that the number of visitors in the past hour would be less than the number of visitors in the past day. This property was derived from the relationship between the changed inputs and changed outputs without knowing anything about how the program operates aside from what the correct functionality should be. So, if a test case was written to verify this property and it failed, the tester would know that the program does not function correctly.

Once MRs are identified and verified to be correct, a test suite is written for the original source code using the MRs to inspire test cases. This suite should run and pass entirely on the source code, verifying that the source code does what is expected. If possible, coverage should be considered during this step as well. However, this suite does not necessarily test specific code functions; instead, it tests overall program functionality. The next step involves mutating the source code, or changing one line in the entire file, based on mutation operators defined for the source code language[1]. For example, in C, a mutant code file may have changed a line from equation (2) to equation (3) below.

```
if(int i > 3)                                                              (2)
if(int i < 3)                                                              (3)
```

The test suite written for the original source code is then run on each mutant code file, of which there are generally thousands depending on the length of the original source file, and assessed for pass rate. If tests fail when running on certain mutants, then those mutants are said to have been killed, as the test suite covered that logical change. If the tests still pass, then those mutants survive. The test suite should be run on the survived mutants multiple times, updating the test suite each time with better test cases. In the end, the goal is to kill every mutant, though generally a score of around 80% is considered sufficient[8].

3 Methodology

Researching encryption algorithms and determining what algorithm to test was the first step of this project. TripleDES looked appropriate due its documentation and structure; there was ample opportunity to describe MRs between input and output properties. Once TripleDES was decided upon for testing, an implementation in Java was searched for, since Java has readily available MT tools such as PIT and MuJava. Metamorphic relationships (MRs) that TripleDES would have were then identified and verified with a series of tests.

3.1 Identified Metamorphic Relationships

The metamorphic relationships used were built from the basic idea that the algorithm should encrypt a plaintext to a ciphertext using a key, and vice versa. TripleDES does this three times to obtain the final ciphertext. The first relationship verified the property that using different keys should lead to a created ciphertext decrypting to different plaintexts. For example, if ’test’ encrypts to ’12345’ using the key ’FFFFF’, then ’12345’ should decrypt to ’test’ using the same key. The first MR states that if ’12345’ decrypts to ’test’ using the key ’FFFFF’, then ’12345’ should not decrypt to ’test’ using the key ’FFFFE’.

The second relationship used the property that slightly different ciphers should not decrypt to the original plaintext. That is, if ’12345’ decrypts to ’test’ using the key ’FFFFF’, then the cipher ’12344’ should not decrypt to ’test’ using the same key. Crytographic algorithms like TripleDES are designed so that similar ciphers will actually decrypt to similar plaintexts, and this vulnerability forms the basis for password cracking. Unfortunately, this may be a design flaw that is necessary to algorithms functioning in their current state. Even so, similar ciphers should not decrypt to identical plaintexts.

The third relationship was the reverse of the first: using different keys to encrypt a plaintext should not result in a similar cipher. If ’test’ encrypts to ’12345’ using the key ’FFFFF’, then ’test’ should not encrypt to ’12345’ using the key ’EFFFF’. The three relationships outlined here formed the basis for a group of ten tests each, which took each MR and modified it for individual test cases for a total of 30 tests. The test suite can be examined in the source code, which is linked below.

Validation of these properties came from first discussing them and determining whether they made sense intuitively, and then testing them through manual input/output relationship examination and the test suite. Another MR was added, discussed at the end of the next section, which increased the test suite coverage slightly and was validated in a similar fashion.

3.2 Testing Setup and Process

For MT to work, all tests should first pass on the original source code. The tests are then assessed on mutated versions of the source code - versions where one line is changed - and if tests fail, then the mutant is said to be killed. If the test suite is perfect, then all mutants should be killed by the tests, but this is generally difficult to achieve. Mutation was not done manually: it was planned to use MuJava, which runs from the command line and creates class level mutants; and PIT, which runs as an Eclipse IDE plugin and creates bytecode level mutants. However, during the initial phase of writing tests for the Java implementation source code, tests kept failing. Most likely this was something to do with how the tests were written, but it was not possible to determine what was causing them to fail, as the failures did not appear to have a pattern. Testing and verifying that TripleDES works correctly within the Java.cipher class is a topic of further research. Because of these difficulties, an implementation of TripleDES in C was desired for testing. The modified source code for the Java implementation can be found at https://github.com/Brick7Face/TripleDES. The original source code for the C implementation can be found on Sourceforge[3].

Finding an implementation of TripleDES in C was not difficult, but finding a mutation engine was. The testing environment also changed from a Windows 10 environment to an Ubuntu 18.04 Linux environment to be able to run the C code and tests more easily from the command line. The new implementation was done completely within two .c files, relying little on outside libraries. The main file was adapted to be a test driver class, which outputted results of execution to a text file and was then verified with a simple Python script. All modified project source code is available at https://github.com/Brick7Face/TripleDES_C. After searching for and testing different mutation engines, one called MUSIC was decided upon, available on Github[16]. Running this created 6227 mutants, which were tested against the same test suite that was modified from the Java implementation to work with the C implementation.

Three rounds of tests were executed: the first round established a baseline and determined which mutants did not compile correctly or had runtime errors, which were thrown out; the second round assessed the results of an improved test suite with a new metamorphic relation; and for the third round the test suite was simply improved by expanding test coverage. The new metamorphic relation verified the property that encrypting a plaintext should result in the same cipher every time that plaintext is encrypted; the cipher should not change if the plaintext is encrypted more than once using the same key. This makes sense intuitively, as a changing encryption process would leave the original plaintext impossible to retrieve. The results of this process are discussed in the next section.

4 Data

Testing during the first round resulted in 1526 of 6627 mutants thrown out due to infinite loops or runtime errors. There ended up being 2043 mutants killed of the remaining 5101, or about 40%. 3058 survived - the other 60%. Before starting the second round of tests, the mutant outputs were compared to the original program output to find the survived mutants that performed identically to the original. These were filtered out and allowed to survive the entire testing process, as these mutations would not be able to be caught by metamorphic test cases if the outputs were identical. Those filtered totaled 1107 of 3058 survived, or around 22% of the total mutants that were not thrown out.

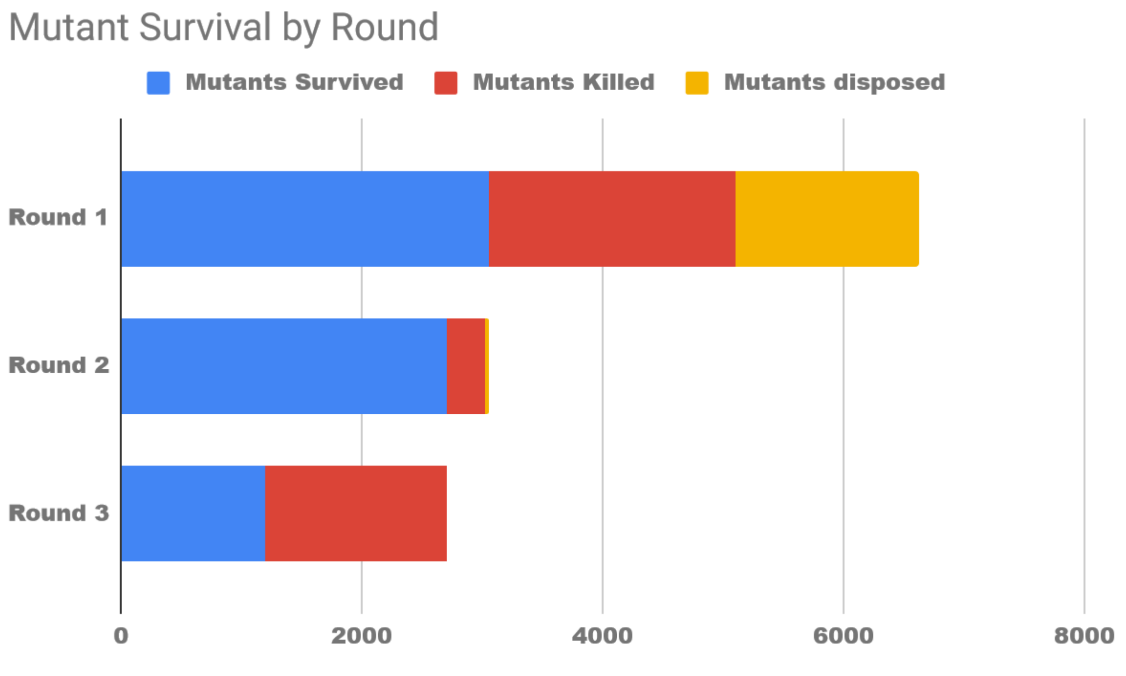

Figure 1. Proportions of mutants disposed, killed, and survived by testing round

The test suite was then updated with the fourth MR described at the end of the previous section, adding 5 more test cases to the suite. The second round resulted in an additional 332 of 3058 previously survived mutants killed and 28 more thrown out. 2698 survived a second time. The total mutant score after this round was around 47%, so adding the fourth MR improved the score around 6-7%.

For the third round, the test suite's coverage was assessed and improved by one line in a function in the tdes.c class. This line made sure that input length was correct and had been absent in previous tests, so an additional test was added to specifically target that line. Adding this single test resulted in 1498 more mutants killed, bringing the final mutant score to 3873 of 5073 killed, or about 76%, yielding an improvement of about 29%. These results are shown visually above in Figure 1.

5 Analysis and Results Discussion

The mutant score was not expected to be very high after the first round of testing, as it was an experiment to determine a baseline for further test improvement. Throwing out the mutants that did not work served to improve the score, as there was a smaller total, and it helped to speed up the following test rounds. 40% was, all things considered, a little lower than expected, but not out of the ordinary for a first test suite. Subsequent testing would further improve the score, with a score of approximately 80% being the end goal.

The second round was more disappointing; though it did improve the score, it was not significant. At this point, it seemed that more MRs would not help, so the coverage of the current MRs was revisited. As seen in Figure 2 below, the coverage improvement made a large difference in the third round. This brought the score closer to what was aimed for. If coverage had been a factor taken into consideration before the mutation step, the results could have been improved further. There were 487 mutants that survived but affected the print statement in tdes.c, which was never called from the main class. Had this been known beforehand, that function would have been removed or modified to be more accessible from the main class; in its current form, it does not work with the testing framework that was built for the rest of the program. That being the case, the overall mutant score could have been increased to a maximum of about 84%. Of course, had mutants been generated after removing the print function, the mutants would have affected other parts of the code, and it is impossible to speculate as to how the tests would have performed given that scenario. Aside from the print function, there was full coverage of the program under test. This was an oversight that affected the results, and so this project could be repeated with that accounted for to obtain a more precise measurement of these MRs on TripleDES.

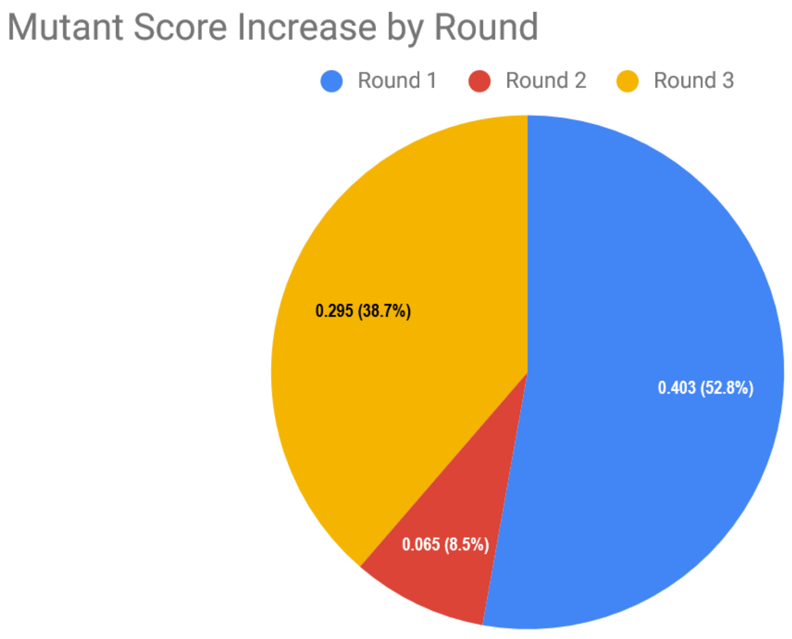

Figure 2. Portions of total mutant score (0.76 = 100%) broken down by round

Those acquainted with software testing know that errors happen more often than not in the 5% of cases not tested, so MT still has some room to improve before it is an airtight testing method. It could serve as a good testing technique to use in tandem with other techniques, as no one approach should be relied upon for thorough testing. The metamorphic relationships used here proved to be effective for TripleDES, and conceptually carried across the two implementations. Even the best MRs cannot make up for issues of coverage, though, so both need to be considered.

Given the fact that cryptography algorithms are inherently difficult to test using a conventional testing technique, MT is a simple way to test a majority of an algorithm with few tests. Whereas it may have taken a large and specific test suite to cover the same amount of code that was covered here, it took less than 40 test cases to cover all of the code aside from the print function, with the added benefit of extra functionality testing. MT serves as a way to identify test cases that may not be touched on by conventional testing techniques, and code changes that a software developer may not have considered.

6 Conclusion

Due to metamorphic testing being a relatively new testing technique, there are few tools freely available to do it effectively. The tool used here, MUSIC, was adequate for exploratory purposes, but could be improved further. Though TripleDES is an end-of-life cryptography algorithm, the results here help demonstrate the effectiveness of MT for cryptography algorithms. As other research has suggested, it is a useful testing technique for programs where the exact output is not known, which is the case for cryptography algorithms.

Further research on this subject can be conducted. This project could be repeated with coverage taken into consideration before mutating the source code. An investigation could be made into the performance of TripleDES in the java.cipher class. A similar project could also be executed using a completely different cryptography algorithm. Metamorphic testing has potential to be more widely leveraged if it is found to be sufficiently useful, and research thus far has suggested this is the case.

7 Acknowledgements

The author wishes to express his gratitude to Dr. Upulee Kanewala for guiding this project. Without her direction, this would have been a lost cause. She had researched MT before and knew what kinds of things to watch for, so that helped in leading to a meaningful outcome.

8 References

1. H. Agrawal et al. Design of Mutant Operators For The C Programming Language. 1999.
2. T.Y. Chen et al. Metamorphic Testing for Cybersecurity. *Computer (Long Beach California)*, 6(49):48-55, 2016.
3. C TripleDES Implementation. https://sourceforge.net/projects/easy-triple-des/?source=typ_redirect
4. D. Flanagan. Java TripleDES Implementation, 2008.
https://github.com/MrSlinkyman/CodeTester/blob/master/src/main/java/com/hernandez/rey/crypto/TripleDES.java
5. M. J. Dworkin et al. *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication, 2001.
6. J. Grabbe. The DES Algorithm Illustrated. *Laissez Faire City Times*.
7. M. Hoffman et al. Generating Keys for Encryption and Decryption. 2017.
8. M. Harmon and Y. Jia. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*.
9. Introduction to the SHA-256 hash function. 2017. https://steemit.com/cryptocurrency/@f4tca7/introduction-to-the-sha-256-hash-function
10. S. Karthik and A. Muruganandam. Data Encryption and Decryption by Using Triple DES and Performance Analysis of Crypto System. *International Journal of Scientific and Engineering Research*, 2(11):24-31, 2014.
11. MuJava Mutation Engine for Java. https://cs.gmu.edu/~offutt/mujava/
12. National Institute of Standards and Technology. NIST Cryptographic Algorithm Validation Program: TDES Validation List. https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program/Validation/Validation-List/TDES
13. PIT Mutation Engine for Java. http://pitest.org/
14. B. Schneier et al. Twofish: A 128-Bit Block Cipher. 1998.
15. M. Sanghavi. Implementing DES Algorithm in Java. http://www.pracspedia.com/INS/DES-java.html
16. swtv kaist. MUtation analySIs tool with High Configurability and Extensibility. https://github.com/swtv-kaist/MUSIC

17. National Institute of Standards U.S. Department of Commerce and Technology. *Secure Hash Standard (SHS)*. Federal Information Processing Standards Publication, 2015.
18. C. Veness. SHA-256 Cryptographic Hash Algorithm. 2017.