The Use of Acceptance Test-Driven Development in the Construction of Cryptographic Software

Alexandre Melo Braga[12], Daniela Castilho Schwab[1], and André Luiz Vannucci[1]
[1] Centro de Pesquisa e Desenvolvimento em Telecomunicações (Fundação CPqD)
Campinas, São Paulo, Brazil
[2] Universidade Estadual de Campinas (UNICAMP)
Campinas, São Paulo, Brazil
Email:{ambraga,dschwab,vannucci}@cpqd.com.br

Abstract—**This paper describes a work in progress on the usage of Acceptance Test-Driven Development (ATDD) during the construction of cryptographic software. As cryptography becomes universalized, it is becoming hard to separate good implementation from bad ones. The paper argues that Test Vectors for cryptography can be used as User Stories in Behavior-Driven Development (BDD) and automate ATDD during software development, complementing algorithm's specification, and contributing to augment software reliability and the overall trust in the correctness of cryptographic implementations. The acquired confidence is preserved even after performing program transformations for improvements, such as performance optimization and hardenings.**

Keywords-BDD; TDD; ATDD; User Stories; Security; Test vectors; Cryptography; Assurance.

I. INTRODUCTION

Nowadays, it is a well-accepted idea that the most likely attacks over software-based cryptosystems are against implementation faults and key management failures [2][16][17]. On the other hand, as cryptography becomes universalized, it is becoming difficult to assure that what is implemented in the real world is actually good cryptography.

The objective of this paper is to discuss preliminary results on the understanding of how the concepts of Behavior-Driven Development (BDD) and Acceptance Test-Driven Development (ATDD) can be applied to the construction of cryptographic software, and how these two concepts can increase both the security and overall trust of software that rely on cryptographic implementations. The idea discussed here was experienced during the construction of a cryptographic library for Android devices [1].

Test-Driven Development (TDD) has become very popular in the agile programming community. In any secure software construction, the correctness of basic security functions is of major concern, and must be preserved even in an ever changing environment. The idea behind this text is that, in cryptography implementation, Test Vectors are User Stories formulated as automated acceptance tests, which can be successfully used to validate the implementation of a cryptographic algorithm against a specification, and prepare the room for future code optimizations and hardenings.

Once automated acceptance tests are available for a specification-based implementation of cryptography, further improvements on the source code can take place in order to address industry concerns, such as performance optimizations, power consumption, and security controls against side-channel attacks and other vulnerabilities in source code. Even after all these transformations, acceptance tests preserve trust by giving strong evidence of correctness.

This work has two motivating drivers. The first is an actual need for increasing the confidence in cryptographic algorithm implementations, which are not under the scrutiny of cryptologists. It is a fact of life that cryptologists are scarcely available human resources, and ordinary programmers are not only more available, but less expensive as well. A frequently asked question in industry is whether or not it is possible to produce high quality implementations of good cryptography, even when there is no cryptologist neither writing source code nor deeply inspecting it.

The second is a lack of literature concerning the combination of TDD, ATDD, or BDD and specific security technologies, like cryptography. Today, common usages of TDD are related to general topics in software, such as enterprise applications, mobile code, data-access code, and so on. The authors could not find any reported case of TDD in cryptographic software.

The text is organized as follows. Section II offers background information on related subjects. Section III details the proposed idea. Section IV discusses practical issues of the proposed approach. Section V contains concluding remarks and future work.

II. BACKGROUND AND RELATED WORK

This section offers background on ATDD, BDD, TDD for security, and Test Vectors for cryptography validation.

A. *Acceptance Test-Driven Development*

ATDD is strongly related to BDD, both of them drive TDD, Acceptance Tests, and Unit Test from User Stories. The core idea of TDD was proposed in 2002 [7] and the state of the practice was studied recently [18], with good text books available on the subject [9].

ATDD drives development on the feature level, similarly to TDD in code level with Unit Tests. Acceptance tests act as micro specifications for the desired behavior and functionality of a system. They tell how the system handles certain conditions and inputs and with what kinds of consequences and outputs. The benefits of ATDD are the

Copyright (c) IARIA, 2015.

following: (i) clear definition of "work done", by providing the knowledge of where the development is and of when to stop working; (ii) promotion of trust and commitment, because there's a direct connection between what the customer specifies and what she gets; and (iii) specification by example, when requirements are expressed by comprehensible examples, rather than by complex formulas or ambiguous descriptions. Tests expressed with concrete examples are easier to read, easier to understand, easier to validate, and easier to write [9].

In Acceptance TDD, a requirement is translated into a set of executable tests and then applied to the implementation, which is validated against tests, rather than against the developer's interpretation of a requirement.

1) User Stories

User Stories are a useful, lightweight technique for managing requirements. User Stories are short sentences written with customer's assistance, stating who does what and why. The story is intended to represent a requirement, acting as a promise of a future conversation between the customer and the developer. A story is typically only one sentence long, it is not intended to document the requirement, and it does not substitute actual specifications.

The most common format or template for User Stories contains the name of the story and three phrases: As a *[user role of the system]*, so that *[I can achieve some goal or objective]*, I want to *[perform some task]*. These three phrases resemble a simple desire of users or customers.

Ideally, a User Story can be formulated as acceptance test before code is written. Well written User Stories, that produce good acceptance tests, usually have quality attributes called Specific, Measurable, Achievable, Relevant, and Time boxed (SMART). The SMART attributes mean that it has to be at least a pair of valid input and corresponding output that: (i) is expressed in the language of the domain specialists; (ii) is concise, precise and unambiguous; and (iii) could be tested within a finite (and short) amount of time. As discussed further in this text, cryptographic test vectors comply with all these attributes.

2) Behavior-Driven Design and TDD

Behavior-Driven Design (BDD) is a way to develop User Stories to describe features on computer programs. BDD concentrates on program's behavior, instead of its implementation. In BDD, the development team asks questions about program's behavior, before and during development, to reduce miscommunication. The questions generate requirements written down as simple User Stories. Later on, User Stories become acceptance tests and integration tests of those programs.

The advantages of BDD are that User Stories are expressed in common language for all stakeholders, and make it feasible to write tests before or during coding. This only feature turns debugging time into validation time. The disadvantages of BDD are twofold: (i) continuous contact with customer is difficult to achieve in most software projects and (ii) BDD almost always leads to bad software architecture, thus requiring frequent refactoring of source code. The compliance to standard Application Programming Interfaces (APIs) and algorithm specifications minimize the impact of this disadvantage in cryptographic software.

Test-Driven Development (TDD), or Test-First Development (TFD), is the practice of writing automated Unit Tests for low-level program constructions (e.g., objects) based on simple User Stories. TDD is guided by a sequence of User Stories obtained from the customer or user. On TDD, the supposed result of writing low-level Unit Tests is that only few defects show up during tests.

Advocates of TDD may question the usefulness of Unit Tests in the presence of automated ATDD. Unit Tests are still useful to validate compliance to programming contracts of an API or to the programming dialog of Frameworks, contributing to regression tests when acceptance tests are not effective. That is exactly the case of cryptography implementation, when testing accessory functionality, such as padding schemes, and conformance to APIs are requirements.

B. TDD and Software Security

There are few works relating TDD or ATDD and security [3][10][21][22]. The work of Smith, Williams, and Austin [3] assesses the relative effectiveness of system and unit level testing of web applications to reveal both SQL injection vulnerabilities and error message information leakage vulnerabilities, when used with an iterative test automation practice by a development team.

More recently, three related works [22][21][10] addressed TDD for security testing. First, Kobashi et al [22] proposed a method to validate implementations of security pattern using TDD. In this method, developers specify the threats and vulnerabilities in the target system during an early stage of development, and then the proposed method validates whether the security patterns were properly applied and assessed whether vulnerabilities were resolved.

Then, Yoshizawa et al [10] evolved the previous work by proposing a validation method, using TDD, for security design patterns in the implementation phase of software development. Finally, Kobashi et al [21] implemented their method in a tool called TESEM (Test Driven Secure Modeling Tool), which supports pattern applications by creating a script to execute model testing automatically. During an early development stage, the developer specifies threats and vulnerabilities in the target system, and then TESEM verifies whether the security patterns are properly applied and assesses whether vulnerabilities are resolved.

None of the above mentioned works treat ATDD in the context of cryptographic software development.

C. Test Vectors for Cryptography

Test Vectors have been used in validation of cryptographic implementations for many years, mostly for product certification, post construction. This section describes the validation of cryptographic implementations with Test Vectors during development.

Test Vectors are data sets constructed with the aim of evaluating the correctness of cryptographic implementations, not their security. However, the functional correctness is a

strong prerequisite for security because, in principle, an incorrect implementation is both unreliable and insecure.

In order to make de validation feasible, cryptographic software under evaluation should allow the necessary control over the input parameters needed for testing. For example, the ability to configure or load known values for the variables required for a specific test may be available via an API. Tests cannot be performed if cryptographic software does not allow control over the values of input parameters.

There are publicly available Test Vectors [11][13][14]. A well-known set of vectors is provided by US National Institute of Standards and Technology (NIST) within the Cryptographic Algorithm Validation Program (CAVP) [13]. All validations based on Test Vectors are designed to test compliance with the norms and standards of the specific algorithm being evaluated. Therefore, they are not meant to provide a measure of the security for a particular cryptographic implementation.

Crafted validation tests are designed to detect accidental defects of implementation and operation, and are not designed to detect intentional attempts to misrepresent validation. For example, malicious implementations can be constructed to give the correct answer for a particular set of tests, then passing as a correct implementation, while concealing some other malicious function. Hence, it is a good practice the use of updated, randomly-generated vectors in conjunction with crafted or standard vectors.

It is noteworthy that Test Vectors are constructed using statistical sampling. That is, only a small amount of samples is extracted from the universe of test cases. Therefore, the successful validation implies strong evidence, but not absolute certainty, of correctness for the implementation under evaluation.

In order to exemplify the structure of Test Vectors, this text uses the Advanced Encryption Standard (AES) [12], along with NIST's vectors [8]. The validation of AES covers various operation modes (e.g., ECB, CBC, OFB, CFB1, CFB8, and CFB128). For each mode, three key sizes are selected (128, 192, and 256 bits).

The AES validation consists of three types of test: Known Answer Tests (KAT), Multi-block Message Test (MMT), and Monte Carlo Test (MCT). There are extra vectors for GCM and XTS modes. The KAT test suite tests four algorithm-specific components. For instance, the GFSbox set tests finite field arithmetic, the KeySbox set tests transactions on subkeys, the Variable Key set tests fixed

```
01 @Test
02 public void pkcs5PaddingTest(){
03    byte[] result = new byte[8],
04    input = "AAAA".getBytes();
05    int inputOffset=0, inputLen=4;
06    int totalInputLen=8, blockLength=8;
07    result = Padding.pkcs5Padding(input,
08          inputOffset,inputLen,totalInputLen,
09          blockLength);
10    assertEquals(Util.ByteArrayToHexStr(result),
11                              "4141414104040404");
12 }
```

Figure 1. Unit Test for PKCS#5 padding of size 4 on a 8-byte block.

TABLE I. FOUR AES KAT VECTORS (ENCRYPTION, CBC, 128-BIT KEY).

Vector type and index	Vector value for each parameter
GFSbox test data for CBC #0	KEY = 00000000000000000000000000000000 IV = 00000000000000000000000000000000 PT = f34481ec3cc627bacd5dc3fb08f273e6 CT = 0336763e966d92595a567cc9ce537f5e
VarKey test data for CBC #0	KEY = 80000000000000000000000000000000 IV = 00000000000000000000000000000000 PT = 00000000000000000000000000000000 CT = 0edd33d3c621e546455bd8ba1418bec8
KeySbox test data for CBC #0	KEY = 10a58869d74be5a374cf867cfb473859 IV = 00000000000000000000000000000000 PT = 00000000000000000000000000000000 CT = 6d251e6944b051e04eaa6fb4dbf78465
VarTxt test data for CBC #0	KEY = 00000000000000000000000000000000 IV = 00000000000000000000000000000000 PT = 80000000000000000000000000000000 CT = 3ad78e726c1ec02b7ebfe92b23d9ec34

plaintext against varying keys, and the Variable Text set tests fixed keys against varying plaintext or cipher texts.

The MMT tests are designed to test the ability of the implementation to process input data consisting of many blocks, and require correct implementation of chaining from block to block. Both KAT and MMT are simple comparisons of known values. MCT still performs comparisons of known values of ciphertexts, but the current ciphertext is computed by chaining previously generated ciphertexts as input to new encryptions into a loop. The last ciphertext is then compared to the value of test vector.

Table I shows KAT vectors for AES encryption in CBC mode with a 128-bit key. The table follows NIST's format and contains examples of the four KAT subtypes. IV, PT and CT stand for Initialization Vector, Plain Text, and Cipher Text, respectively.

III. TDD AND ATDD FOR CRYPTOGRAPHY

This section proposes an approach to perform TDD and ATDD over cryptographic implementations. First, it discusses a strategy for conducting Unit Tests that fits on the TDD framework. Then, a strategy to perform ATDD with Test Vectors as User Stories is presented and discussed.

The code snippet in Figure 1 is an example of how JUnit [6], a simple framework to write repeatable tests in Java, can be used in automated testing of security functions. The method tests the structure of padding in PKCS#5 format. The code works for blocks of 8 bytes (for example, used by 3DES) and the input data of 4 bytes, so the function must include 4 bytes of padding with the hexadecimal value 0x4. This code can be generalized to other test cases for padding.

In the case of padding, the exhaustive coverage of all possible test cases becomes feasible, since the padding has a small number of options that depend on the block size of the cryptographic algorithm. For instance, there are 8 test cases for algorithms with 64-bit block, as well as 16 test cases for

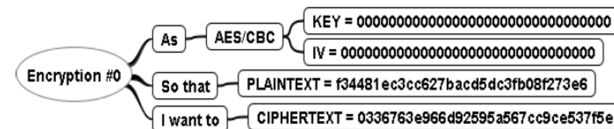

Figure 2. The User Story for a single Test Vector.

Copyright (c) IARIA, 2015.

algorithms with 128-bit blocks. Considering both insertion and removal of padding as distinct test cases, there are 48 test cases. In terms of JUnit, there are two approaches for implementing these test cases. In one, all the 48 test cases can be individually automated. In other, one single function can be built for all test cases. An option sitting in between is to build two separated functions, one for padding additions and other for removals.

The choice of a method for each test case is preferable because complies with TDD's philosophy of identifying errors quickly and directly. Moreover, the option with a single method encapsulating all test cases still implies that there must be debugging to identify which test cases have failed. Therefore, by not eliminating depuration, it yields only partial benefit from TDD philosophy. A similar approach can be adopted to test cryptographic routines, but with major limitations. TDD and JUnit can be used to test the basic operation of the encryption routine by using particular Test Vectors for encryption and decryption.

Despite being useful for simple functional tests of security functions, unit tests (based upon JUnit) do not scale well when applied to ATDD for cryptography. If each possible cipher text is considered a test case, then the amount of test cases, though finite and countable, is incredibly large. Even a small sample, but statistically significant, greatly increases the time to perform unit tests. For this reason, ATDD is most suitable for the validation of cryptographic implementations than simple TDD based upon stand-alone JUnit tests. Figure 2 illustrates the first vector of Table 1 represented as a User Story for AES encryption. This single test case is one in thousands of test cases. For instance, NIST's vectors for AES in CBC mode consist of more than 2,700 single tests.

Figure 3 depicts the overall idea of using ATTD for cryptography. Test vectors are input data to test cases, which are programmed as (automated) User Stories intended for Acceptance Tests. Then, ATDD asserts the expected behavior of cryptographic algorithms, resembling BDD. This approach can be complemented by ordinary unit tests. The point of contact between User Stories and cryptographic implementations is the cryptographic API. In this way, test cases do not act directly on algorithms internals, but verifies its behavior, as seen from the API perspective.

Java programs, such as the one shown in Figure 4, were built to enable ATDD through Java Cryptographic Architecture (JCA) [4][5]. Before being used in a custom cryptographic library [1], the ATDD test suite was validated against two presumed correct implementations of JCA [5][23]. The test suite was used for testing not only pure Java code, but also C code encapsulated by Java Native Interface (JNI) adapters and available through the JCA API.

Figure 4 illustrated how NIST's Monte Carlo Tests (MCT) [8] can be performed by the proposed ATDD test suite. The figure shows the source code for a Java method to perform MCT tests for encryption in ECB mode. This Java code is almost a direct translation of the pseudo code from [8]. This code snippet was meant for AES, but can be generalized for any block cipher, because it does not depend on the specific test data nor cipher implementation. The

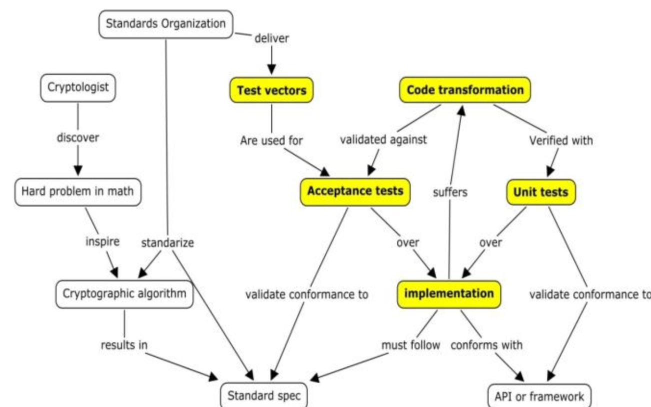

Figure 3. ATTD asserts whether an implementation follows its specified behavior.

cipher function (a wrapper for the actual encryption function) is called only two times, in lines 17 and 19. The loop from line 10 to line 36 computes a chain of ciphertexts, which is saved (in line 25) for future comparisons.

IV. PRACTICAL ISSUES AND DISCUSSION

This section discusses practical considerations that arise when implementing the proposed approach.

```
01  void mcEncECB(byte[] key, byte[] plainText) {
02    byte[][] bK  = new byte[1000][];//WorkingKey
03    byte[][] bPT = new byte[1000][];//PlainText
04    byte[][] bCT = new byte[1000][];//CipherText
05    TestVector vt;
06    calcVT = new TestVector[100];
07    bK[0] = key;
08    bPT[0] = plainText;
09
10    for (int i = 0; i <= 99; i++) {
11      vt = new TestVector();
12      vt.setPT(b2x(bPT[0]));
13      vt.setKey(b2x(bK[i]));
14      int j;
15      for (j = 0; j <= 999; j++) {
16        if (j == 0) {//init cipher with key
17          bCT[j] = crypt(bK[i], bPT[j], true);
18        } else      {//reuse cipher
19          bCT[j] = crypt(bK[i], bPT[j], false);
20        }
21        if (j < 999) { bPT[j+1] = bCT[j];}
22      }
23      j--; // leaves loop when (j == 1000)
24      vt.setCT(b2x(bCT[j]));
25      calcVT[i] = vt;
26      if (i <= 99) {
27        if (key.length == 16) { // 16*8 = 128 bits
28          bK[i+1] = xor128(bK[i], bCT[j]);
29        }
30        if (key.length == 24) { // 24*8 = 192 bits
31          bK[i+1] = xor192(bK[i], bCT[j-1], bCT[j]);
32        }
33        if (key.length == 32) { // 32*8 = 256 bits
34          bK[i+1] = xor256(bK[i], bCT[j-1], bCT[j]);
35        }
36      }
37      bPT[0] = bCT[j];
38   }
39   return;
40  }
```

Figure 4. Monte Carlo Test for encryption in ECB mode, in Java.

A. *Examples of Defects Found*

Defects did occur during development. Thus, in addition to assert compliance to specifications, test cases are structured to detect implementation faults, including problems with pointers, insufficient memory allocation, incorrect treatment of errors, and other incorrect behaviors. This section gives examples of the more interesting defects found, usually not related to simple misunderstandings of specifications.

The proposed method was applied over standard (e.g., AES, RSA, HMAC, and SHA-2) and non-standard (e.g., Serpent, Salsa20, and Blake) cryptography. When testing the implementation of AES, two kinds of failures were identified. Failures in padding were found when zero padding and AnsiX923 padding were used and the value of the last byte was less than 16, resulting in a mismatch in test vectors. A failure in the CTR mode of operation occurred when the first 16 bytes (first block) was correctly encrypted (matched test vector), but the remaining blocks do not. Other failure was caused by a missing initialization of the hash function after the first call to it into HMAC computation.

Non-standard algorithms produced more severe failures than standard ones. It is worth to mention the cases for three algorithms: Serpent, Blake, and Salsa20. When testing a C implementation of Serpent against its test vectors [19], several implementation failures were discovered concerning memory leaks, wrong output for either encryption and decryption, and program crashes after 1,000 iterations.

Salsa20 and Blake lack extensive test vectors. In fact, only a few unofficial tests were found for Salsa20, which were complemented by random tests produced by a reference implementation. Blake was tested only with random tests produced by a reference implementation. A bug was detected in Blake that was caused by wrong calls from Blake224 to functions of Blake256. Also, platform upgrades (from 32 bits to 64 bits), downgrades (from 64 to 32 bits) and changes (from different flavors of Linux) caused errors in encryption and hashes that could be detected by regression tests.

Similar defects were found during development of all cryptographic algorithm implementations. By the time of writing, there were no collected statistics on the efficiency of the proposed method. However, the examples mentioned above suggest the method worth the effort when there is no cryptologist timely available to support debugging.

B. *Lessons Learned*

Standard algorithms produced fewer defects than non-standard ones. That is probably because standard algorithms possess better documentation available to developers as well as good reference implementations.

The Java APIs for symmetric encryption, secure hash functions, and Message Authentication Codes (MAC) have shown good testability (meaning the ease with which a given test coverage criterion can be satisfied [15]) against NIST's vectors. Unfortunately, the same cannot be said for asymmetric encryption. JCA was created before the advent of TDD and ATDD, and is not testable by design. The authors have found that parts of the API presented poor testability and are not suitable for testing with NIST's vectors. In particular, the API for asymmetric encryption was designed with "textbook" RSA in mind, and does not allow for RSA-PSS and RSA-OAEP to be easily tested, because it's not possible to setup some of the parameters required by NIST for these two randomized algorithms. Similarly, the key agreement API was designed with "textbook" Diffie-Hellman (DH) in mind and suffers from the same issue when used with Authenticated DH implemented according to NIST's specifications. This means that, in order to be fully testable, an implementation has to sacrifice conformance to JCA API. Also, security issues have been found in JCA [16].

Concerning API compliance, a lesson learned is that any cryptographic implementation should have the ability of being tested by third parties. Co-design of both functional code and test code can favor testability. But that may not the case when testing legacy cryptography with ATDD.

TDD and ATDD can in fact reduce the time of debugging, when looking for the causes of failures. A lesson learned is that TDD uses the same techniques of debugging, but in a productive way, writing automated tests during software development.

Failure isolation seems to be the most important advantage of TDD to development of cryptographic software. In TDD, if a test fails, the cause must be the most recently added code. Failure isolation is almost trivial in TDD, because at any moment, all previous test cases must have passed. Modes of operation as well as the internals of cryptographic implementations are hard to debug otherwise.

Usually, good test cases are not sufficient for TDD to produce good code. Also, good design is eventually accomplished by code refactoring. In the case of standard cryptographic algorithms, there will always be specifications and reference implementations. Furthermore, conformance to an API (e.g., JCA), minimizes the need for refactorization.

Finally, the most criticized disadvantage of TDD is that it is strongly dependent on tester's experience to produce good test cases. In cryptography, there is a concern about the need for an oracle that could provide good-enough test cases. According to [15], an oracle is any (human or mechanical) agent that decides whether a program behaved correctly in a given test and accordingly gives a verdict of pass or fail. In case of cryptography, that need is satisfied by test cases provided either by cryptologists or standards organizations.

Unfortunately, non-standard algorithms usually lack test cases and can only benefit from relatively small test sets supplied by their authors or other practitioners. In this work, for both standard and non-standard cryptography, reference implementations were used as oracles for generation of random test cases, in complement of third-part test vectors.

C. *Test Vectors as Metrics for Quality Measures*

In order to be useful as a quality measure, tests must have clearly defined meanings for success and failure. This text adopted the meanings from the Software Engineering Body of Knowledge (SWEBOK) [15] as follows: a fault is the (root) cause of a malfunction and a failure is the undesired effect observed in the behavior of programs. Thus, testing can reveal failures, but it is the faults that can and must be removed from programs. Still [15], the generic term defect

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-427-5

can refer to either a fault or a failure. The result of testing a single cryptographic implementation with its test vectors is twofold: passing them all constitutes success with a justified confidence, but failing only once reveals a failure and compromises the whole implementation.

Failed test cases can be used as indirect measures of how distant an implementation is from achieve conformance to that test cases and can be used for estimation of effort, team assignment, overall cost estimation, and influencing how long a test effort should be continued. In this work, the criteria for test termination could be positively defined by passing all test cases. The term "passed them all" was directly related to how much testing was enough and when a test period could be concluded. It also involved concerns about costs and risks incurred by possible remaining failures, as opposed to costs incurred by continuing to test.

Additionally, when considering whole cryptographic libraries with implementations for various algorithms, to make testing more effective in making quality predictions, it is important to know which types of failures may be found and the relative frequency with which these failures have occurred in the past. The failure density for an implementation under test can be evaluated by counting discovered failures as the ratio between the number of failures found and the size of that implementation. This evaluation was left as a future work.

V. CONCLUDING REMARKS

This paper argues that Test Vectors are User Stories and can automate acceptance tests for cryptographic software. Test Vectors are good acceptance tests because they meet halfway between cryptologists and developers. They are User Stories from the problem domain, that don't look like source code, providing an easy way to reach agreement. The approach presented in this text increases confidence in cryptographic software by maintaining a strong evidence of correctness, even after many code transformations.

Future work includes the use of ATDD in other cryptographic algorithms and protocols. Further research includes the design of customized Test Vectors. In order to be useful, metrics and statistics concerning the efficiency of the approach still have to be collected in structured ways. Finally, studies have to be done to combine the approach with methods for secure software development.

ACKNOWLEDGMENT

The authors acknowledge the financial support given to this work, under the project "Security Technologies for Mobile Environments – TSAM", granted by the Fund for Technological Development of Telecommunications – FUNTTEL – of the Brazilian Ministry of Communications, through Agreement Nr. 01.11.0028.00 with the Financier of Studies and Projects - FINEP/MCTI.

REFERENCES

[1] A. Braga and E. Morais, "Implementation Issues in the Construction of Standard and Non-Standard Cryptography on Android Devices," in proc. of the 8th International Conf. on Emerging Security Information, Systems and Technologies (SECURWARE), 2014, pp. 144–150.

[2] B. Schneier, Security in the Real World: How to Evaluate Security Technology, Comp. Sec. Journal, 1999, vol.15, n. 4.

[3] B. Smith, L. Williams, and A. Austin, "Idea: using system level testing for revealing SQL injection-related error message information leaks," *in* Proc. of the 2nd International Conference on Engineering Secure Software and Systems (ESSoS), 2010, pp. 192-200.

[4] Java Cryptography Architecture (JCA) Reference Guide. [retrieved: July, 2015] docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html.

[5] Java Cryptography Architecture, Providers Documentation for JavaSE 7. [retrieved: July, 2015] docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html.

[6] JUnit 4. [retrieved: July, 2015] http://junit.org.

[7] K. Beck, Test Driven Development By Example, Addison-Wesley Longman, 2002.

[8] L. E. Bassham III. The Advanced Encryption Standard Algorithm Validation Suite (AESAVS). National Institute of Standards and Technology, 2002.

[9] L. Koskela, Test Driven: Practical TDD and Acceptance TDD for Java Developers, 2007.

[10] M. Yoshizawa, T. Kobashi, H. Washizaki, Y. Fukazawa, T. Okubo, H. Kaiya, and N. Yoshioka, "Verifying Implementation of Security Design Patterns Using a Test Template," in proc. of the 9th Int. Conf. on Availability, Reliability and Security (ARES), 2014, pp. 178–183.

[11] NESSIE Test Vectors. [retrieved: July, 2015] www.cosic.esat.kuleuven.be/nessie/testvectors.

[12] NIST FIPS-PUB-197, Announcing the Advanced Encryption Standard (AES), FIPS Publication 197, 2001.

[13] NIST. Cryptographic Algorithm Validation Program (CAVP). [retrieved: July, 2015] csrc.nist.gov/groups/STM/cavp.

[14] OpenSSL Validation Suite. [retrieved: July, 2015] opensslfoundation.com/testing/validation-2.0/testvectors.

[15] P. Bourque and R. Fairley, Eds., Guide to the Software Engineering Body of Knowledge (SWEBOK), ver 3. IEEE Comp. Society, 2014.

[16] P. Gutmann, "Lessons Learned in Implementing and Deploying Crypto Software," in Proc. of the 11th USENIX Security Symposium, Dan Boneh (Ed.), 2002, pp. 315-325.

[17] R. J. Anderson, "Why cryptosystems fail," Communications of the ACM, vol. 37, n. 11, 1994, pp.32-40.

[18] S. Hammond and D. Umphress, "Test driven development: the state of the practice," in proc. of the 50th Annual Southeast Regional Conf., 2012, pp. 158-163.

[19] Serpent - A New Block Cipher Proposal for AES. [retrieved: July, 2015] www.cs.technion.ac.il/~biham/Reports/ Serpent.

[20] Snuffle 2005: the Salsa20 encryption function. [retrieved: July, 2015] http://cr.yp.to/salsa20.html.

[21] T. Kobashi, M. Yoshizawa, H. Washizaki, Y. Fukazawa, N. Yoshioka, T. Okubo, and H. Kaiya, "TESEM: A Tool for Verifying Security Design Pattern Applications by Model Testing," in proc. of the IEEE 8th Int. Conf. on Software Testing, Verification and Validation (ICST) , 2015, pp. 1–8.

[22] T. Kobashi, N. Yoshioka, T. Okubo, H. Kaiya, H. Washizaki, and Y. Fukazawa, "Validating Security Design Patterns Application Using Model Testing," in proc. of 8th Int. Conf. on Avail., Reliability and Security (ARES), 2013, pp. 62–71.

[23] The Legion of the Bouncy Castle. [retrieved: July, 2015] www.bouncycastle.org/java.html.