HAL
open science

Learning and Grading Cryptology via Automated Test Driven Software Development

Konstantin Knorr

To cite this version:

Konstantin Knorr. Learning and Grading Cryptology via Automated Test Driven Software Development. 13th IFIP World Conference on Information Security Education (WISE), Sep 2020, Maribor, Slovenia. pp.3-17, 10.1007/978-3-030-59291-2_1 . hal-03380704

HAL Id: hal-03380704

https://inria.hal.science/hal-03380704v1

Submitted on 15 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Learning and Grading Cryptology via Automated Test Driven Software Development

Konstantin Knorr

Trier University of Applied Sciences, Germany
knorr@hochschule-trier.de

Abstract. Understanding common cryptological concepts like encryption, hashing, signatures, and certificates is a prerequisite when working as an IT security professional but it is also a major challenge in security education. Often students struggle with cryptology as sound previous mathematical knowledge is required and study time is limited. Teachers face the problem to fairly assess the students' knowledge and understanding of cryptology. The paper presents an approach to face these challenges by utilizing test driven software development techniques for students who have taken courses in programming and theoretical cryptology. The paper describes the practical experience gained in courses with ~30 students utilizing a specialized client-server system to automate the tests. We propose that this setup is beneficial for learning as it gives immediate feedback and allows students to focus on the erroneous parts of their software. The test cases can also be used to grade students' code by weighting the test cases e.g. in an exam setting.

Keywords: Cryptology, Java, JUnit Tests, Test driven Software Development, Playfair Cipher

1 Introduction

Students learning cryptology face the challenge to understand complex topics like number theory for asymmetric ciphers in a limited time frame. Cryptology is typically taught and exercised on a pen and paper basis in theory only due to an overloaded curriculum. This dilemma is additionally fostered by new developments in cryptology like elliptic curves and post quantum techniques which require even more theory and time.

We argue in this paper that learning cryptology in combination with test driven software development (TDSD) is beneficial for the students and also allows for a transparent grading system providing quality assessment of students. TDSD provides instant feedback, supports learning form failures, is programming language independent, and gives an immediate evaluation. Grading the student's source code by predefined test cases relieves the instructor from evaluating the code which is tedious, difficult and error-prone. Software testing is also a major step towards the greater goal of software quality. Edwards and Perez-Quinones [1] argue that despite the importance of the topic, most computer science curricula provide only minimal coverage of the topic, as it is poorly suited for the topic of a new course. Students therefore typically leave university unprepared for real world testing tasks upon graduation.

In many universities the curriculum for computer science encompasses courses for programming and for IT security which often include theoretical cryptology. The scope is given in Europe following the Bologna reform in ECTS (European Credit Transfer System) credit points (CP), one CP equals ~30 working hours. In many computer science study programs, the IT security scope is limited to 5-10 CPs. This does not leave enough time for an in-depth coverage of cryptology. However, by creating a new course e.g. called "Implementing Cryptology", based on previous skills in cryptology and programming, the understanding in both areas can be improved.

Courses typically consist of lectures where the theoretical content is presented by the lecturer and exercises in which work assignments for the theoretical content have to be solved by the students (homework). Most modules need to be graded by the lecturer. The grade is typically the result of a written exam or an oral examination. Some universities' examination regulations allow for requiring students to have successfully solved a certain percentage of the assignments prior to taking the exam or oral examination. While the exam is typically in a highly supervised environment (cheating will lead to "failed"), the homework is typically done in an unsupervised environment e.g. at home. One of the major challenges for the lecturer is to find fair and transparent grades for the students while taking his time effort into consideration especially for courses with a large number of students.

The paper describes the LCJTC (Learning Cryptology with Java Test Cases) approach which has been applied to courses with 20-30 students. We argue that LCJTC allows students to better learn and understand cryptology and also provides a transparent and fair grading schema.

The remainder of the paper has the following structure: Section 2 gives the necessary background information on TDSD, JUnit tests, the system used to assess the students' code, and cryptology in Java. As an easy-to-understand example for a cryptological cipher we will use the Playfair cipher presented in Section 2.4 throughout the paper. Section 3 describes the LCJTC approach including grading. The paper closes in Section 4 with a discussion of LCJTC in practice including the topics cheating, privacy issues, and related work. Finally, a conclusion including future work is given.

2 Background

2.1 Test Driven Software Development & JUnit Test

TDSD aka test-first coding stems from Beck [2] and is a subgroup of agile software development. The fundamental idea is to produce only code that passes predefined test cases. Coding first and then writing test cases could result in omitting important test cases. It encourages programmers to always have a running version of their code and to test features and code during implementation thereby avoiding integration problems typically encountered in other development models at the end of a project.

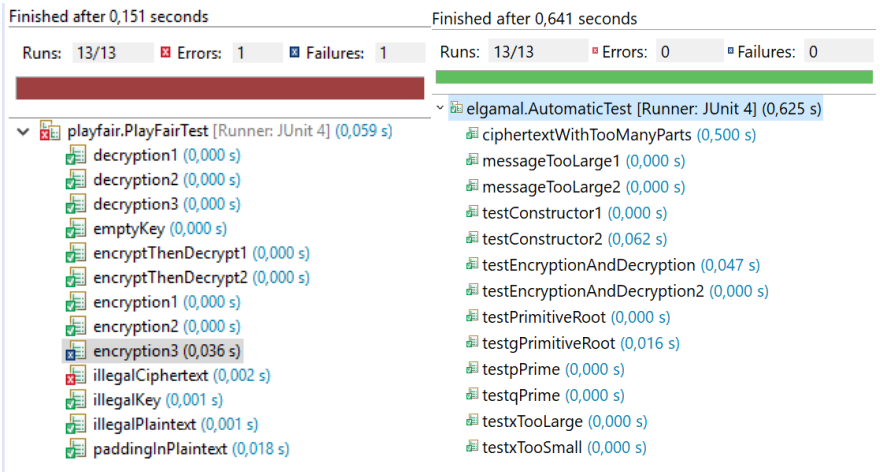

Figure 1. JUnit test cases for Playfair (left) and Elgamal (right) ciphers and their run time in the Eclipse IDE. For Playfair 11 of 13 tests passed. Test `encryption3` produced a failure, as an incorrect ciphertext was calculated, test `illegalCiphertext` raised an unexpected exception. All Elgamal tests passed.

A programmer writes a test and then the code that needs to be tested. The test is a piece of software, too and is typically written in the same IDE. When another programmer later wants to make changes to the code, he first executes all test cases to ensure that the code base is error free. He then changes or adds code, again executes all test cases, and makes sure they pass in order not to add any errors in this new code. In case tests do not pass, he knows that he introduced an error and can fix it. This cycle is repeated until all test cases pass. Summarized, the major steps in TDSD are: 1. add a test, 2. run all tests, 3. write some code, 4. run tests, 5. refactor code, 6. repeat from step 1.

TDSD is programming language independent. It is mainly applied to unit tests in comparison to the later and more complex integration, system, and acceptance tests. Several so called XUnit frameworks have been proposed for unit tests. For the Java programming language, the popular JUnit framework [3] implements the TDSD paradigm and allows for testing classes, methods, and exceptions. Major IDEs like Eclipse provide graphical user interfaces to visualize JUnit tests and their results (cf. Fig. 1). A JUnit test can have three different results:

- Passed (coloured green), e.g. test `decrpytion1` in Fig. 1.
- Error (coloured red): an unexpected exception has been thrown by the code.
- Failure (coloured blue): an incorrect result has been calculated for that test.

2.2 Automatic Software Evaluation System

Automatic assessment and grading of programming assignments has been in the focus of researchers and practitioners for many years. See [1, 4-6] for an introduction and overview of current systems. The main advantages of these systems are traceability of

the programming progress, immediate feedback, and scalability. Especially for larger cohorts automated grading is a vital tool in study programs.

This paper makes use of the existing and long-running ASE (Automatic Software Evaluation) system which has been described in detail in [7, 8]. ASE has been in use since 2006 at the University of Applied Sciences in Trier, Germany. Its architecture is a web based client server system. Authentication of students and instructors is accomplished via Shibboleth login. ASE's kernel comprises web sockets for managing client-server communication and database and execution environments for selected programming languages like Java, C++, and Python. Each language can load plugins e.g. unit test runners or modules to check programming style.

The different tests can be configured using XML files. This includes maximal time lengths allowed for the tests. The result of the assessment of the student's code is stored in a database and written in an XML file which is then presented in the student's browser. The ability to assess graphical Java programs written in JavaFX even though the server has no desktop environment installed differentiates ASE from other systems. ASE uses the TestFX framework ("headless tests") for this purpose. ASE also uses TLS to protect all data in transit. A typical life cycle for ASE assignments is the following:

- The instructor presents theoretical teaching content in class and prepares corresponding assignments (s. Appendix A for an example), a time frame for the assignment, the test cases and uploads the test cases to the server.
- Students solve the assignments and submit the solution to ASE.
- ASE verifies the submission by executing the test cases on the submitted code and provides instantaneous feedback which parts are correct or wrong with error messages that are part of the test cases (e.g. line 6 in Listing 2). Also, peer-review is possible allowing students to review each other's code.
- After the deadline the instructor reviews assignment results and can comment on common errors in the next lecture and optionally makes adaptions to the test cases and assignment text.

2.3 Java & Cryptology

Java by Oracle is a popular object-oriented programming language especially in the academic world. It is based on classes with attributes and methods including constructors. Classes can be grouped into packages. Java also allows for an elaborate exception handling. The JCA (Java Cryptography Architecture) [9] encompasses hash functions, key generation, is part of the java.security package, and was introduced with JDK 1.1. JCE (Java Cryptography Extension) is the Java interface for encryption, decryption, and authentication and is included in the package javax.crypto [10]. It is part of Java since JDK 1.4. The Cipher class forms the core of the JCE framework, providing the functionality for encryption and decryption.

As encryption for symmetric key sizes larger 56 bit and for asymmetric key sizes larger 512 bit underlies export restrictions in the US, JCA does not contain any encryption operations and all encryption is handled in JCE allowing only "weak" keys in default installation. The local Java security policy needs to be replaced (for version pre

1.8.0_151) or edited (for newer versions) for larger key sizes, otherwise an InvalidKey exception is thrown. For many cryptological scenarios JCA and JCE are not sufficient and external cryptology providers like Bouncy Castle [11] are required. External providers typically have (1) more cipher suites and algorithms than the default JCE, (2) other helpful utilities e.g. for encoding and decoding or for reading arcane formats like PEM, and (3) are of non US origin and are therefore not subject to the rigid US export restrictions and possible legal consequences.

```
1.  import javax.crypto.Cipher;
2.  import javax.crypto.SecretKey;
3.  import javax.crypto.spec.*;
4.  import org.bouncycastle.util.Strings;
5.  import org.bouncycastle.util.encoders.Hex;
6.  public class gcmEncryptor {
7.  public static void main(String[] args) throws Exception{
8.  SecretKey aesKey = new SecretKeySpec(
        Hex.decode("000102030405060708090a0b0c0d0e0f"), "AES");
9.  byte[] iv = Hex.decode("001122334455667788 99aabb");
10. byte[] msg = Strings.toByteArray("Hello Maribor!");
11. System.out.println("msg:   " + Hex.toHexString(msg));
12. Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BC");
13. GCMParameterSpec spec = new GCMParameterSpec(64, iv);
14. cipher.init(Cipher.ENCRYPT_MODE, aesKey, spec);
15. byte[] cText = cipher.doFinal(msg);
16. System.out.println("cText: " + Hex.toHexString(cText));}}
```

Listing 1. Java code to encrypt the string “Hello world!” with AES in GCM mode.

Listing 1 shows the most important steps when using modern symmetric ciphers with Java. It uses a hard-coded key which should never be done for productive code. This is done here only for the sake of simplicity. Also, the optional addition of authenticated data and the decryption is not shown.

The Cipher class is instantiated in line 12 with the symmetric cipher AES in GCM[1] mode which does not require padding. Note that the external crypto provider Bouncy Castle (“BC” in line 12) is used. The initialization of the encryption in line 14 requires an AES key which is generated in line 8 (hardcoded, 128 bit) using the SecretKey class and a specification object for GCM mode (cf. line 13, 64-bit tag and 96 bit initialization vector). Following these preparatory steps, the message is encrypted using the doFinal method in line 15 and printed to screen in line 16. Note that all cryptological functions operate on bytes in Java. Therefore, in order to encrypt a string in Java it needs to be converted in bytes (cf. line 10). To print bytes to screen the Bouncy Castle Hex class is used. Most of the objects can raise exceptions e.g. when an incorrect key length is used (cf. line 7).

[1] Galois Counter Mode = modern block mode for symmetric ciphers like AES which allows for encryption and authentication of data.

Listing 1 shows the complexity of implementing modern cryptology in Java. Several classes need to be used together in a correct and safe way, e.g. Cipher, SecretKey, SecretKeySpec, and GCMParameterSpec. The complexity further increases when using asymmetric cryptology as two different keys (public and private) are used. JCE also offers so called factory classes – another complex topic typically not covered in introductory programming classes. This stresses the importance of teaching cryptology in combination with software engineering. Otherwise students can produce vulnerable code [12].

Table 1. Playfair square for the passphrase CRYPTOLOGY

C	R	Y	P	T
O	L	G	A	B
D	E	F	H	I
K	M	N	Q	S
U	V	W	X	Z

2.4 Playfair Cipher

For illustration purposes we introduce the Playfair cipher. Playfair is a historic cipher and can easily be broken with computer aid e.g. by frequency analysis. Nevertheless, we use it here for didactic purposes as an easy to understand example.

The Playfair cipher is named after Lord Lyon Playfair and was invented by Sir Charles Wheatstone in 1854. It is a substitution cipher which was used in WWI [13] and uses the alphabet = {A, …, I, K, L, ..., Z}. J is not used and is replaced by I in the plaintext and key. Double characters in the plaintext are separated by an X. If the length of the resulting plaintext is uneven, an additional X is added at the end (padding). The plaintext is then split in two character blocks ("bigrams"). For the plaintext EDUCATION the following bigrams are processed: ED, UC, AT, IO, NX. Based on the key which is a string a so called 5x5 Playfair square is generated. Assume the key equals CRYPTOLOGY. The key letters are written in the square starting top left continuing to the right. Double letters (like O and Y in our key) are omitted. The remaining letters are inserted in alphabetical order. The resulting Playfair square for our key is given in Table 1.

To encrypt, bigrams will be mapped to bigrams using the Playfair square. (1) If both plaintext letters are in the same row, replace them with their right neighbours (in the example: ED => FE). Start from left, when the right neighbour is outside the square. (2) If both plaintext letters are in the same column replace them with their lower neighbours (UC => CO). Take the top entry when leaving the square. (3) In all other cases replace them with the letters on the same row respectively but at the other pair of corners of the rectangle defined by the original pair (AT => BP). The resulting ciphertext is FECOBPDBQW. Note that the rules require different letters which is guaranteed by the padding.

Decryption is very similar to encryption. As Playfair is a symmetric cipher, the receiver needs the same key using a secure channel. To decrypt it, he first generates the

same Playfair square using the key. Secondly, he takes bigrams of the ciphertext and applies the following three rules to them: in encryption rule (1) replace right by left, in (2) lower by upper. Rule (3) is identical. Decrypting the ciphertext yields EDUCATIONX. Note the terminal X which is due to padding.

3 Learning Cryptology with Java Test Cases

The combination of using JUnit test cases to teach students cryptology and also grade their code will be called LCJTC (Learning Cryptology with Java Test Cases) from now on. The approach has been used in practice in the Bachelor course "Applied Cryptology with Java" at Trier University of Applied Sciences, Germany. The course is worth 5 CPs and is an optional course in the computer science curriculum. The course was given in the winter semesters 2016/17 and 2018/19. In the first course 35 students participated in the assignments and 26 took the exam. In the second course 25 students worked on the assignments and 20 took the exam. The mandatory use of ASE (cf. Section 2.2) was introduced in 2018/19. As a prerequisite for participating in the course students had to pass classes in (1) IT security (5 CPs) including a theoretical introduction to cryptology and in (2) object oriented programming in Java (10 CPs). Classes were held on a weekly basis over a period of 14 weeks with 90 minutes of lecture and 90 minutes for the assignments per week.

3.1 Assignments and Their Selection

Table 2 shows the content of the lectures and the corresponding assignments. The Playfair assignment is given in Appendix A as a sample. The topics covered roughly follow standard cryptological text books like [14] for the theoretical part and [15, 16] for Java cryptology. The assignments were chosen based on the recommendations given in [12, 17, 18] and selected for didactical or practical purposes. The TLS assignment e.g. was chosen due to the paramount importance of the TLS protocol to protect TCP traffic. The PasswordSafe assignment illustrates the challenges to be tackled when generating and encrypting credentials. CryptoTimer illustrates that asymmetric ciphers encrypt much slower than symmetric ones (by a factor of 10-1000 in our setting). The BMPEncryptor e.g. illustrates the shortcoming of using the ECB mode when encrypting bitmap images.

Several assignments like CertValidator, CipherExceptor require an inverse way of thinking about cryptology. Contrary to other assignments where correct code must be developed these assignments require to deliberately raise exceptions, produce errors, or break encryption and provide test cases to illustrate this behaviour. This inverse way of thinking nicely illustrates typical problems in cryptological software and hopefully prevents students from making these errors in the future [12, 17].

Assignments marked with * in Table 2 were assessed by ASE. The source code of the other assignments had to be turned in via an e-learning platform. For selected assignments the code had to be presented to the instructor during class. The last four weeks of the course were used for student projects where 1-3 students worked on self-

selected projects, cf. Lecture 10 for sample topics. The projects were presented in class in the last two weeks of the course.

Table 2. Topics of lectures and assignments

	Content of Lecture	Assignments
1	JUnit-Tests, exception handling, introduction to ASE, classical substitution and transposition ciphers	1. Playfair cipher*, 2. Railfence cipher*
2	JCA & JCE, Bouncy Castle, modern symmetrical block ciphers including modes of operation and padding	1. BMPEncryptor which shows differences of ECB vs. CTR encryption, 2. CipherExceptor raises all exceptions contained in Java's Cipher object
3	Stream ciphers, hash functions, MAC, PBKDF, pseudo random functions	1. PasswordSafe, 2. PasswordHasher: hashing and storing passwords in /etc/shadow-style, 3. cracking a stream cipher via timing attack, 4. CBC-MAC
4	Java's BigInteger class, asymmetric cryptology: RSA, Rabin, Elgamal	1. BigInteger, 2. RSA cipher*, 3. Rabin cipher*
5	Homomorphic Encryption, Paillier crypto system	1. Elgamal cipher*, 2. Paillier cipher*, 3. CryptoTimer compares run times of ciphers
6	Digital Signatures incl. RSA, DSA, ECDSA, Merkle, and Lamport signatures	1. JavaSigner: signing files with RSA, DSA or ECDSA, 2. Lamport signatures*
7	Java Keystore, X.509 certificates	1. KeyStoreReader* (reading and using credentials from a given keystore), 2. X509Printer (printing X.509 details to screen), 3. CertValidator: create JUnit test cases for selected certificate exceptions
8	Transport layer security (TLS) protocol	1. Build up a TLS connection using a given certificate chain and credentials
9	Elliptic curve cryptology	1. MyFirstEllipticCurve*, 2. ECDH in Java
10	Shamir's secret sharing, selected advanced cryptological topics like post quantum ciphers, e-voting, e-cash, cryptological card games	1. Shamir Secret Sharer*

Implementing cryptological software is subtle, can easily lead to attackable code, and should be done by professionals. Notorious problems are side channel attacks e.g. when implementing RSA or insufficient pseudo randomness. Therefore, a central paradigm is to teach students not to code their own crypto code in practical settings but to use existing and well established code like OpenSSL or Bouncy Castle [11] for scenarios with more than low security requirements. Nevertheless, the assignments include cipher implementation like RSA, Elgamal, or Playfair for didactical purposes.

3.2 Cryptological JUnit Tests

Cryptological concepts comprise but are not limited to encryption/decryption, signature and signature verification, hashing, pseudo random number generation [14]. Each of

these concepts can be further grouped into test categories, e.g. for symmetric encryption schemas like Playfair the following categories can be defined:

- C1: Correct encryption: Given a plaintext and key, check if the correct ciphertext is calculated.
- C2: Correct decryption: Given a ciphertext and key, check if the correct plaintext is calculated.
- C3: Decryption as inversion of encryption: Using the same key, is the encrypted and then decrypted plaintext equal to the original plaintext?[2]
- C4: Invalid inputs, parameters, and padding: (1) invalid input or encoding for keys, plaintext or ciphertext, (2) invalid parameters like wrong key length, wrong block size, wrong length of an initialization vector, and (3) invalid padding

```
1.  public class PlayFairTest extends TestCase{
2.  @Test
3.  @TestOrder(10)
4.  @TestDescription("Check correct encryption of Playfair cipher")
5.  @DependsOnCorrectnessOf("SignatureTests.testSignature")
6.  @TestFailureMsg("The correctly encrypted text should be" +
        "FECOBPDBQW")
7.  public void encryption1() {
8.  PlayFair pf = new PlayFair("CRYPTOLOGY", "EDUCATION", true);
9.  String ciphertext = pf.getCiphertext();
10. Assert.assertEquals("Your implementation returns <code>" + cipher-
        text + "</code>.", "FECOBPDBQW", ciphertext);}
11. @Test
12. @TestOrder(11)
13. @TestDescription("Check illegal keys")
14. @DependsOnCorrectnessOf("SignatureTests.testSignature")
15. @TestFailureMsg("Only characters a-z or A-Z are allowed as keys.")
16. @Test(expected = IllegalCharacterException.class)
17. public void illegalKey() throws IllegalCharacterException {
18. new PlayFair("CRYPTO*LOGY", "SUPPER", true);}}
```

Listing 2. ASE Playfair test case `encryption1` checks the correct encryption of a given plaintext, test case `illegalKey` checks if an `IllegalCharacterException` is thrown when an illegal key is used.

For asymmetric ciphers and signature schemas additional categories exist. Typically, private and public keys need to fulfil certain number theoretical preconditions and are dependent upon each other. Private keys are used for different operations compared to public keys, see the Elgamal tests in Fig. 1 for examples. Implementing tests for categories C1-C3 is straightforward in Java. Test case `encryption1` defined in lines 2-10 in Listing 2 is an example for C1. These tests are implemented using assertions like

[2] Note that some encryption schemes like Playfair in combination with certain padding schemas can yield different decrypted ciphertexts than the original plaintext.

`Assert.assertEquals` in line 10 provided by the JUnit framework. C4 tests require a different approach and can be tested by requiring a certain exception (cf. line 16 in Listing 2). Test `illegalKey` defined in lines 11-18 is an example. Note that no assertion is necessary. The constructor in line 18 raises the `IllegalCharacterException` required in line 16. The assignment in Appendix A requires this exception for characters different from A-Z. The key in line 18 contains the illegal character *.

Listing 2 contains several ASE features. The annotation `@TestOrder` allows to define a mandatory chronological order of test cases, `@TestDescription` allows to provide a description of the test case that will be presented in the ASE GUI to the students. `@DependsOnCorrectnessOf` is used to check if other tests have successfully passed before it. The signature tests listed in lines 5 and 14 check if the naming conventions for package, class, and methods required in the assignment have been followed by the students. Finally, `@TestFailureMsg` provides the opportunity to display a specific message in case of a failed test. See [7, 8] for more details.

Table 3. Grading students A, B and C with selected weighted Playfair test cases

Playfair Test Cases	Category	Weight	Student A	Student B	Student C
encryption1	C1	2	1	1	0
encryption2	C1	2	1	1	0
decryption1	C2	2	0	1	0
decryption2	C2	2	0	1	0
emptyKey	C4	1	1	1	1
illegalKey	C4	1	1	1	1
illegalPlaintext	C4	1	1	1	1
illegalCiphertext	C4	1	1	1	1
encryptThenDecrypt1	C3	3	0	1	0
encryptThenDecrypt2	C3	3	0	1	0
paddingInPlaintext	C4	1	1	1	1
Score			8	19	5

3.3 Grading

Especially for larger cohorts automated and fair grading of student software code is desirable. Following the LCJTC approach this is done in the following way: Prior to be admitted to the exam, students had to (1) pass ~75% of the test cases in ASE, (2) present ~75% of the non-ASE assignments to their instructor and (3) present and document their student projects. As students are used to JUnit test cases, these are also used in the exam setting. Depending on the complexity of the test case and the category, an additional weight factor is applied.

Table 3 illustrates the approach following the Playfair cipher. Test cases from all four categories C1-C4 are used. C3 test cases have weight 3, C1 and C2 ones have weight 2, and C4 tests have weight 1. The sample results of three students are shown. Test cases are rated on a 1/0 basis (pass => 1, fail =>0). The score of the Playfair assignment is the weighted sum of the rated test cases. Scores for the different exam assignments can be summed up to yield the final grade for the student. Student B in Table 3 passed all tests and got the best score, while student C only managed to solve C4 test cases.

4 Discussion and Conclusion

4.1 Discussion

Students typically spent several hours spread over 3-4 days on each assignment. They tended to prefer assignments which used existing Java ciphers over coding their own ciphers like Rabin. Most of the students were able to solve the assignments. Exam time had to be extended from 90 to 120 minutes as understanding the assignments took the students longer than expected. It is planned to hand out a skeleton implementation which includes some classes and methods to speed up the coding. The grading time of the exam is apart from smaller preparatory steps almost instantaneous.

Feedback on LCJTC by the students has been extremely positive, with students expressing clear appreciation for the practical benefits of TDSD. In the questionnaires filled out after the course, the students additionally indicated an increased interest in cryptology after the course. Several students however disliked coding their own ciphers (like Playfair and RSA) and preferred to use existing implementations. The difficulty of the assignments was rated OK by 75% and “Difficult” by the remaining 25%. Over 90% of the students liked the final student project. The main critique was to drop the final exam and generate the grade based on the weekly assignments. This is planned for future classes but requires changes to the examination regulations of the course.

JUnit tests have also been used for grading the exams of the students. The students were given only the assignments and not the test cases. The grades have been calculated as described in Section 3.3. Students had the opportunity to inspect their code after the exam. The test cases have been valuable in showing the students their errors. The transparency and fairness of the approach has been positively acknowledged.

Handing out the test cases also would allow a specific way of cheating for categories C1 and C2. As an example consider test case `encryption1` in Listing 2. The test case contains the plaintext and corresponding ciphertext. The student could easily write the following encryption method that passes this test without understanding the encryption rules of the Playfair cipher. Pseudocode: `If (plaintext == EDUCATION and key == CRYPTOLOGY) then ciphertext = FECOBPDBQW`. Of course, this is easily detected by manual review but not by an automatic assessment. In future exams two disjunct sets of test cases can be used: one for the exam, the second for grading. This calls for a test case generator.

Automatically testing student code on ASE raises several privacy related questions. Students need to authenticate to ASE and each of their commits and the corresponding

results are stored in a central database. A lot of privacy related information is evident or can be deduced. Examples are the percentage of passed and failed tests, their preferred working time, and sociological aspects like with whom they work on their assignment. In Europe, the General Data Protection Regulation gives clear guidelines for personal data. The use of ASE is currently voluntary. Students not wanting to use ASE will be given the necessary test cases and can demonstrate the passing of the test cases directly to the instructor. A more complex legal issue arises when enforcing the use of ASE for assignments or exams and the student is not willing to give his consent.

When using ASE for the first time, students need to agree on the privacy notice which also explains the purpose and usage of the data collected, their rights (e.g. deletion of their data), and time frames for storing the data. ASE's only purpose is to check the committed code against the tests. Internally, pseudonyms instead of student names are used. After a course is finished all corresponding data is automatically erased.

4.2 Related Work

The idea of using TDSD in the classroom is not revolutionary. Desai et al. [19] provide statistical evidence proving the benefit of TDSD in academia. Especially automated grading has raised the interest of many researchers, see [1, 4-6] but not exclusively for cryptographic software. Isong [5] describes an approach typical of such grading systems. Her automated program checker focuses on compiling and executing student programs against instructor-provided test cases, and then assigns a grade based on the comparison of the actual output of the student program against expected results provided by the instructor – an approach very similar to ASE. Edwards and Perez-Quinones [1] propose to let students pass in their code plus their JUnit test cases. Grading can then be done by submitting the student code to all passed-in test cases complemented by master test cases provided by the instructor.

Braga et al. [20] use TDSD in the construction of cryptographic software. They argue that test cases can automate acceptance tests for cryptographic software. Cryptographic test cases are considered good acceptance tests as they meet halfway between cryptologists and developers. However, they do not use Java and use a non-educational setting.

Testing software with fuzzing is a well-established technique when testing the robustness of software. Fuzzing involves providing invalid, unexpected, or random data as inputs [21]. The software is then monitored for exceptions such as crashes or potential memory leaks which falls into category C4 defined above and does not take into account C1-C3. Fuzzing has found lots of cryptographic failures e.g. the notorious TLS heartbleed attack. A more recent example is given in [22].

To the best knowledge of the author this is the first application of TDSD techniques to learn and grade software for cryptography in an educational setting.

4.3 Conclusion and Future Work

Coding cryptology is beneficial for understanding and learning cryptology. Only when the concepts are fully understood, it can be implemented correctly. And only then the test cases will pass. Sticking to the test cases also helps to split a large assignment into

smaller, more conquerable parts. In our Playfair example, the steps could e.g. be padding, input sanitization, creation of the Playfair square, encryption, and decryption (cf. Section 2.4 and Appendix A).

For instructors LCJTC represents a major time shift from correcting students' code ex post to creating sample solutions, test cases, and detailed assignments prior to classes. The positive effect of this approach is that it is scalable and can thereby be applied to larger cohorts and is also well-suited for distance learning. The time effort for instructors can be reduced by sharing resources with other instructors[3], thereby extending the range and number of assignments. Possible extensions include post-quantum ciphers [23] or side-channel and timing attacks.

As Listing 1 clearly indicates, using modern ciphers in Java is a complex task. As ASE already supports other programming languages like Python or C++ it is planned to explore crypto assignments in other languages. This approach can then be used to compare crypto implementations e.g. concerning the time needed for encryption.

For some assignments like CertValidator, CipherExceptor students are required to hand in their own test cases, following the classical TDSD paradigm as described in Section 2.1. However, in many assignments the instructor defines the test cases and codes a reference solution while the student tries to write code that passes the given test cases. This is due to ASE's current inability to test test cases, which should be addressed in future ASE versions.

References

1. Edwards, S. Pérez-Quiñones, M.: Experiences using test-driven development with an automated grader. Journal of Computing Sciences in Colleges 22(3), 44-50 (2007).
2. Beck, K.: Test-Driven Development: By Example. Addison Wesley (2002).
3. JUnit Homepage, https://junit.org, last accessed 2020/1/29.
4. Iffländer, L. et al.: PABS – a Programming Assignment Feedback System. In: Proc. of the 2. Workshop Automatische Bewertung von Programmieraufgaben (2015).
5. Isong, J.: Developing an automated program checker. J. Computing in Small Colleges, 16, 3, 218-224 (2001).
6. Krusche, S., Seitz, A.: ArTEMiS - An Automatic Assessment Management System for Interactive Learning. SIGCSE 2018, February 21-24, Baltimore, MD, USA (2018).
7. Herres, B., Oechsle, R., Schuster, D.: Der Grader ASB. In: "Automatisierte Bewertung in der Programmierausbildung", Herausgeber Oliver J. et al., Waxmann-Verlag, pp. 255-271 (2017).
8. Schuster, D. et al.: Automatische Bewertung von JavaFX-Anwendungen. In: Proc. of the 3. Workshop Automatische Bewertung von Programmieraufgaben (2017).
9. Knudsen, K.: Java Cryptography, O´Reilly (1998).
10. Weiss, J.: Java Cryptography Extensions. 1st edn. Morgan Kaufmann (2004).
11. Bouncy Castle Homepage, https://www.bouncycastle.org, last accessed 2020/3/12.
12. Lazar, D. et al.: Why does cryptographic software fail? A case study and open problems. In: Proc. of 5th Asia-Pacific Workshop on Systems, pp. 1-7. (2014).

[3] If you are interested, please contact the author. Note that the material is currently only available in German.

13. Kahn, D.: The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet. Scribner (1996).
14. Stinson, D., Paterson, M.: Cryptography: Theory and Practice. 4th edn. CRC (2018).
15. Hook, D.: Beginning Cryptography with Java. Wrox (2005).
16. Hook, D., Eaves, J.: Java Cryptography: Tools and Techniques, ebook, https://leanpub.com/javacryptotoolsandtech (2020).
17. Long, F. et al.: The CERT Oracle Secure Coding Standard for Java. Addison-Wesley (2011).
18. McGraw, G.: Software Security – Building Security In. Addison-Wesley (2006).
19. Desai, C., Janzen, D., Savage, K: A survey of evidence for test-driven development in academia. ACM SIGCSE Bulletin, June (2008).
20. Braga, A., Schwab, D., Vannucci, A.: The Use of Acceptance Test-Driven Development in the Construction of Cryptographic Software, 9th Int. Conference on Emerging Security Information, Systems and Technologies (2015).
21. Takanen, A., DeMott, J., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House (2008).
22. Aumasson, J., Romaillerm, Y.: Automated testing of crypto software using differential fuzzing. Blackhat Conference, US (2017).
23. NIST Homepage for Post-Quantum Cryptography, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography, last accessed 2020/2/9.

Appendix A: Playfair Assignment

In order to illustrate the LCJTC method, we will use the Playfair cipher introduced in Section 2.4 and show a corresponding assignment here:

Implement the Playfair cipher in Java like introduced in Section 2.4 with the following requirements:

- Package name = playfair
- Class name = PlayfairCipher
- Attributes of class: `private char[][] charTable = new char[5][5]`. Contains the Playfair square, `String key, String plaintext, String ciphertext`
- Methods:
  - Constructor: `PlayfairCipher(String key, String text, boolean encrypt)`. Boolean encrypt indicates, if the text is encrypted (value=true) or decrypted (value=false). The constructor sets all attributes to their correct value.
  - `void createTable(String key)`. Generates the Playfair square given a key
  - `String encrypt(String plaintext)`. Encrypts plaintext.
  - `String decrypt(String ciphertext)`. Decrypt ciphertext.
  - `String prepareText(String text)`. Padding and replacing J with I.
- For plaintexts, ciphertexts and keys containing illegal characters an IllegalCharacterException must be thrown.
- Test vector: plaintext = EDUCATION, key = CRYPTOLOGY, ciphertext = FECOBPDBQW.