CogniCrypt: Supporting Developers in Using Cryptography

Stefan Krüger*, Sarah Nadi†, Michael Reif‡, Karim Ali†, Mira Mezini‡, Eric Bodden*,
Florian Göpfert‡, Felix Günther‡, Christian Weinert‡, Daniel Demmler‡, Ram Kamath‡
*Paderborn University, Germany, {fistname.lastname}@uni-paderborn.de
†University of Alberta, Canada, {nadi, karim.ali}@ualberta.ca
‡Technische Universität Darmstadt, Germany, {reif, mezini, guenther, weinert, demmler}@cs.tu-darmstadt.de,
fgoepfert@cdc.informatik.tu-darmstadt.de, aramachandrakamath@gmail.com

***Abstract*—Previous research suggests that developers often struggle using low-level cryptographic APIs and, as a result, produce insecure code. When asked, developers desire, among other things, more tool support to help them use such APIs. In this paper, we present CogniCrypt, a tool that supports developers with the use of cryptographic APIs. CogniCrypt assists the developer in two ways. First, for a number of common cryptographic tasks, CogniCrypt generates code that implements the respective task in a secure manner. Currently, CogniCrypt supports tasks such as data encryption, communication over secure channels, and long-term archiving. Second, CogniCrypt continuously runs static analyses in the background to ensure a secure integration of the generated code into the developer's workspace. This video demo showcases the main features of CogniCrypt: youtube.com/watch?v=JUq5mRHfAWY.**

***Index Terms*-Cryptography, Code Generation, Variability Modeling, Code Analysis**

I. Introduction

Cryptography is the primary means of protecting sensitive data on digital devices from eavesdropping or forgery. For this protection to be effective, the used cryptographic algorithms must be conceptually secure, implemented correctly, and used securely in the respective application. Despite the availability of mature and (still-)secure-to-use cryptographic algorithms, several studies have indicated that application developers struggle with using the Application Programming Interfaces (APIs) of libraries that implement these algorithms. For example, Lazar et al. [14] investigated 269 cryptography-related vulnerabilities and found that only 17% are related to faulty implementations of algorithms, while 83% result from application developers misusing cryptographic APIs. Other studies suggest that approximately 90% of applications using cryptographic APIs contain at least one misuse [6, 9].

To investigate the reasons for this widespread misuse, we previously triangulated the results of four empirical studies, one of which was a survey of Java developers who previously used cryptographic APIs [19]. Our results show that the majority of participants found the respective APIs hard to use. When asked what would help them use these APIs, they suggested better documentation, different API designs, and additional tool support. In terms of API design, participants used terms like *use cases*, *task-based*, and *high-level*. These suggestions indicate that developers struggle with the fact that cryptographic APIs reside on the low level of cryptographic algorithms instead of being more functionality-oriented APIs that provide convenient methods such as `encryptFile()`. When it comes to tool support, participants suggested tools like a *CryptoDebugger*, *analysis tools* that find misuses and provide *code templates* or *generate code* for common functionality. These suggestions indicate that participants not only lack the domain knowledge, but also struggle with APIs themselves and how to use them.

In this paper, we present CogniCrypt, an Eclipse plugin that enables easier use of cryptographic APIs. In previous work, we outlined an early vision for CogniCrypt [2]. We have now implemented a prototype of the tool that currently supports developers in the following ways:

- Generate secure implementations for common programming tasks that involve cryptography (e.g., data encryption).
- Analyze developer code and generate alerts for misuses of cryptographic APIs.

II. CogniCrypt in a Nutshell

We will use the cryptographic task "Encrypt data using a secret key" (ENC) as a running example throughout the paper. When an application developer - CogniCrypt's user - selects this task, CogniCrypt generates code that implements a simple encryption using Java's `Cipher` API.

Figures 1 and 2 illustrate the steps that a user must follow in CogniCrypt for ENC. First, to trigger the code generation, the user clicks on the CogniCrypt button in Eclipse's tool bar. The dialog shown in Figure 1 then pops up and the user has to select both the target project for code generation and ENC from the list of supported tasks. The user then answers a few high-level questions that do not require deep cryptography knowledge. The answers to these questions help CogniCrypt generate the appropriate source code. One such question for ENC is "Should your key be derived from a user-specified password?". Once the user has answered all questions for ENC, CogniCrypt presents the user with a list of algorithms (or combinations thereof) in different configurations and auto-selects the most secure one as shown in Figure 2. The user may change the selection through a drop-down menu. For the more keen user, CogniCrypt provides more detailed information about the selection in the same window. After the user hits

978-1-5386-2684-9/17/$31.00 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Tool Demonstrations

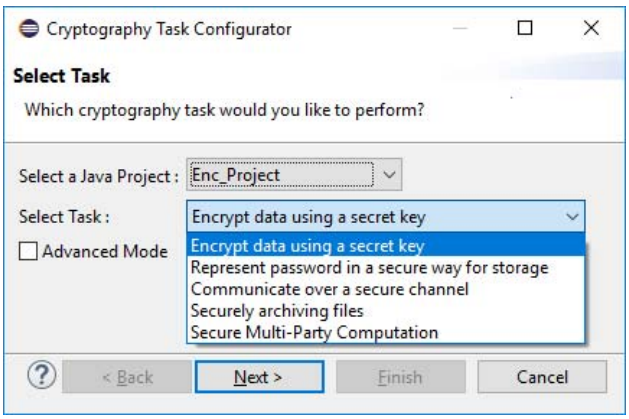

Fig. 1. Dialog for task selection.

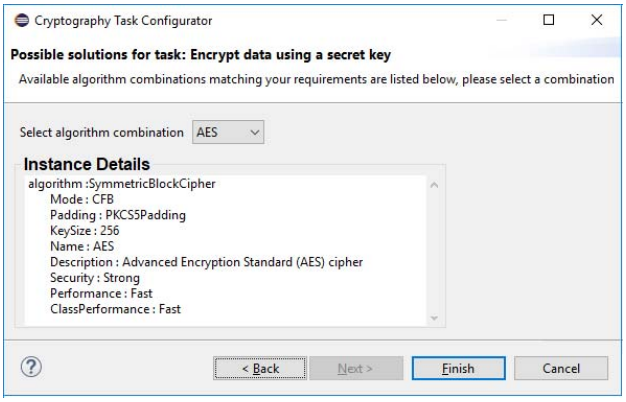

Fig. 2. Algorithm selection screen.

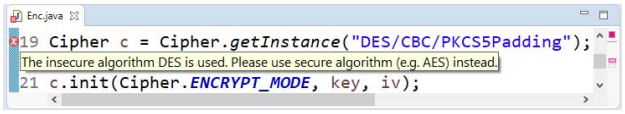

Fig. 3. Reporting misuses as Eclipse error markers.

the finish button, CogniCrypt generates two code artefacts into the user's Java project: the code that implements ENC into a package named `crypto`, and a method that demonstrates how the user may use the generated code in their own project.

In addition to code generation, CogniCrypt notifies the user of misuses of cryptographic APIs by running a static-analysis suite automatically every time the code is compiled. CogniCrypt generates an Eclipse error marker for each detected misuse of the supported cryptographic APIs. Figure 3 depicts a warning issued by CogniCrypt when the user changes the generated code for ENC to use the insecure encryption algorithm DES.

III. Generating Secure Code

CogniCrypt's code-generation component enables it to generate secure implementations for several cryptographic programming tasks. Each task in CogniCrypt is specified using three artefacts: a model describing the involved algorithms, the task's implementation, which CogniCrypt may provide the user with, and a code snippet demonstrating its usage. Figure 5 illustrates the general workflow and necessary artefacts. We will refer to different parts of the elements in Figure 5 using the circled numbers shown in the figure (e.g., ①).

A. Modelling Cryptographic Algorithms in Clafer

The cryptography domain comprises a wide range of algorithms, each can be configured in multiple ways. This *variability* becomes an issue for developers with little or no experience in cryptography, because not all algorithms and configurations are secure to use in every context. Therefore, developers have to figure out which algorithm may be used in which situation. To help developers close this knowledge gap, we systematize the domain knowledge by means of a variability model ① using Clafer [3], a variability modelling language that facilitates a mix of class and feature modelling. Clafer supports two constraint solvers that can *instantiate* a model, i.e., generate instances of the model that satisfy all its constraints. Any element in Clafer is called a *clafer* and can either be abstract or concrete. The difference is that the instance generator does not create instances for abstract clafers. In prior work [18], we describe our modelling approach and discuss the trade-offs of using other variability modelling languages.

Figure 4 shows a simplified version of the Clafer model for ENC in CogniCrypt. The model defines the abstract clafer `Algorithm` in Line 1. Lines 2–4 define the attributes of `Algorithm` (`name`, `security`, and `performance`). The model defines three abstract clafers that extend `Algorithm` (i.e., inherit its attributes): `SymmetricBlockCipher` (Line 7), `KeyDerivationalgorithm` (Line 15), and `Digest` (Line 24). Each extension defines additional attributes. Moreover, `SymmetricBlockCipher` defines two constraints (Lines 11–12). The model then defines concrete clafers for all ENC-related cryptographic algorithms by extending the three `Algorithm`-type clafers (Lines 26–56). Finally, the clafer definition for ENC (Lines 61–64) includes all its necessary cryptographic algorithms such as a symmetric block cipher (Line 64). If the user decides to derive the key from a password, the definition is updated to require a key derivation algorithm.

B. Configuring a Solution

The generated code for each task is specified as an XSL-based code template ⑤ to enable code generation by an XSL transformation. Figure 6 depicts an excerpt of the stylesheet for ENC representing part of its implementation. The code implements a simple encryption bootstrapped with an initialization vector. Since each task may be implemented in multiple ways, the stylesheet may contain one or more *variability points*, that is statements that depend on the configuration of task. ENC has one variability point: the argument to the call `Cipher.getInstance()` (Lines 77–82). The class `Cipher` is used for encrypting data, and the argument to `getInstance()` specifies the encryption algorithm, block mode, and padding scheme of the encryption [21, Section on class `Cipher`].

To generate valid code, CogniCrypt resolves this variability by asking the user questions to help it configure a solution ②.

```
abstract Algorithm
 name -> string
 security -> integer
 performance -> integer

abstract SymmetricBlockCipher
 : Algorithm
 keySize -> integer
 mode -> Mode
 padding -> Padding
 [mode != ECB]
 [padding != NoPadding]

abstract KeyDerivationAlgorithm
 : Algorithm
 iterations -> integer
 outputSize -> integer
 digest -> Digest?
 [outputSize =128 || outputSize
  =192 || outputSize =256]
 [digest.security != Broken]
 [iterations =1000]

abstract Digest : Algorithm
   outputSize -> integer

AES: SymmetricBlockCipher
 [name = "AES"]
 [keySize =128 || keySize =192
   || keySize= 256]
 [keySize =128 => performance =
  VeryFast && security =Medium]
 [keySize > 128 => performance =
  Fast && security =Strong]

DES: SymmetricBlockCipher
 [name = "DES"]
 [performance = VeryFast]
 [security = Broken]
 [keySize =56]

SHA_1: Digest
 [name = "SHA-1"]
 [performance = VeryFast]
 [security = Weak]
 [outputSize =160]

SHA_256: Digest
 [name = "SHA-256"]
 [performance = Fast]
 [security = Strong]
 [outputSize =256]

pbkdf2 : KeyDerivationAlgorithm
 [name = "PBKDF2"]
 [performance = Slow]
 [digest]
 [security = digest.security]

abstract Task
 description -> string

PasswordBasedEncryption : Task
 [description = "Encrypt data using
  a given password"]
 cipher -> SymmetricBlockCipher
```

Fig. 4. Clafer model for the password-based encryption (ENC) programming task.

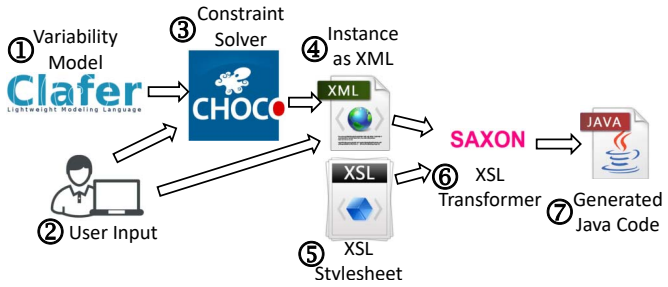

Fig. 5. The workflow of code generation in CogniCrypt.

For the supported currently tasks, the authors have developed these questions. The task selection determines the parts of the stylesheet that are relevant to the user and the questions that will be presented to the user. For example, for ENC, the user may choose to derive the key from a password. If they do so, CogniCrypt automatically modifies the Clafer model such that a key derivation algorithm is also required to implement the task, not only a symmetric block cipher as shown in Figure 4. In general, each answer either adds constraints to the Clafer model (e.g., setting a security level of the cipher to *high* or *very high*) or influences the generated code directly (e.g., by adding or removing a call or changing a parameter value).

After answering all questions, CogniCrypt runs the constraint solver Choco [15] on the Clafer model to generate all its instances, one of which is a version of the model with all variability resolved ③. For the ENC model in Figure 4, there are at least three distinct instances with different values (128, 192, 256) for the `keysize` attribute of `AES`. The result is a list of combinations of algorithms in different configurations that CogniCrypt shows to the user in the final dialog sorted by security level in descending order. CogniCrypt automatically selects the first solution, but the user can change the selection.

C. Generating Code

CogniCrypt stores the selected configuration in an XML file ④. The code is then generated into the user's project under the package `crypto` by performing an XSL transformation using SAXON [24] bootstrapped with the stylesheet and the XML configuration file ⑥. Two code artefacts ⑦ are generated: the code implementing the task, and a method that demonstrates how the developer can use the implementation. This method is usually generated into an extra class in the same package as the generated implementation code. In case a Java file from the target project is currently opened in the editor, CogniCrypt generates the method into this file. It also ensures the method is generated within the respective class, but outside existing methods. With the XSL stylesheet in Figure 6 and the configuration from Figure 2 as inputs, CogniCrypt generates the class `Enc` in Figure 7. The developer may choose to keep the generated code as is or integrate it into their application code in a different way.

IV. Enforcing Secure Implementations

In addition to generating code, CogniCrypt continuously applies a suite of static analyses to the developer's project in the background. These analyses ensure that all usages of cryptographic APIs remain secure, even when the developer modifies the generated code for better integration into their project or to add some functionality. Moreover, if the developer uses the cryptographic APIs directly (i.e., without using the code-generation component), running the analysis suite ensures secure usage of the APIs.

To statically analyze the underlying Eclipse project, CogniCrypt uses TS4J [4], a fluent interface in Java that defines and evaluates typestate analyses [26]. A typestate analysis helps CogniCrypt determine the set of allowed operations in a specific context. TS4J is implemented as an Eclipse plugin on top of the static analysis framework Soot [27]. CogniCrypt reports misuses by generating error markers directly on the left gutter within the Eclipse IDE as shown in Figure 3.

```
<xsl:if test="//task/algorithm[@type='
 SymmetricBlockCipher']">
<xsl:result-document href="Enc.java">
package <xsl:value-of select="//task/Package"/>;
<xsl:apply-templates select="//Import"/>
public class Enc {
 public byte[] enc(byte[] data, SecretKey key){
  byte [] ivb = new byte [16];
  SecureRandom.getInstanceStrong().nextBytes(ivb);
  IvParameterSpec iv = new IvParameterSpec(ivb);
  Cipher c = Cipher.getInstance("
   <xsl:value-of select="//task/algorithm[@type='
    SymmetricBlockCipher']/name"/>/
   <xsl:value-of select="//task/algorithm[@type='
    SymmetricBlockCipher']/mode"/>/
   <xsl:value-of select="//task/algorithm[@type='
    SymmetricBlockCipher']/padding"/>");

  c.init(Cipher.ENCRYPT_MODE, key, iv);
  byte [] res = c.doFinal(data);
  byte [] ret = new byte[res.length + ivb.length];
  System.arraycopy(ivb, 0, ret, 0, ivb.length);
  System.arraycopy(res, 0, ret, ivb.length, res.length);
  return ret;
 }
}
</xsl:result-document>
</xsl:if>
```

Fig. 6. XSL stylesheet for ENC.

```
public class Enc {
 public byte[] enc(byte[] data, SecretKey key) {
  byte[] ivb = new byte[16];
  SecureRandom.getInstanceStrong().nextBytes(ivb);
  IvParameterSpc iv = new IvParameterSpec(ivb);

  Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
  c.init(Cipher.ENCRYPT_MODE, key, iv);
  byte [] res = c.doFinal(data);
  byte [] ret = new byte[res.length + ivb.length];
  System.arraycopy(ivb, 0, ret, 0, ivb.length);
  System.arraycopy(res, 0, ret, ivb.length, res.length);
  return ret;
 }
}
```

Fig. 7. Generated code for ENC.

Figure 8 depicts one of the TS4J rules used in CogniCrypt to detect the usage of the outdated encryption algorithm DES when instantiating a `Cipher` object (e.g., Line 100 in Figure 7). The first part of the rule (Line 119) tracks the parameter of calls to `Cipher.getInstance()` (as defined in Line 109) and binds it to the temporary variable `CIPH`. The rule also stores the statement in variable `GETINS` for later reference. After the `orElse()` keyword (Line 121), the rule checks whether the value bound to `CIPH` equals `"DES"`. If that is the case, CogniCrypt reports the error `"DES is used"` at the statement stored in `GETINS` as Figure 3 shows.

V. Integrated Components

CogniCrypt provides support for cryptographic APIs in a task-based manner. In this section, we discuss the tasks that CogniCrypt currently supports, providing short conceptual descriptions, how they are implemented, and the decisions users have to make in the tool. We also discuss the cryptographic algorithms that CogniCrypt supports.

```
109  String CIPHER_CALL =  "<javax.crypto.Cipher:
110  javax.crypto.Cipher getInstance(java.lang.String)>";
111
112  enum Var { CIPH };
113  enum StmtID {GETINS}
114
115  protected Done<Var, State, StmtID> atCallToReturn(
116  AtCallToReturn<Var, State,  StmtID> d) {
117
118  return d.atCallTo(CIPHER_CALL).
119           always().trackParameter(0).as(CIPH).
120           and().storeStmtAs(GETINS).
121           orElse().atAssignTo(CIPH).
122           ifValueBoundTo(CIPH).startsWith("DES").
123           reportError("DES is used").atStmt(GETINS);
124 }
```

Fig. 8. TS4J Rule to find usages of DES.

A. Supported Tasks

1) Symmetric Encryption :

a) Description: encryption of data as a byte array.

b) Implementation: implementations of symmetric block ciphers in SunJCE Provider [23] such as AES, Triple-DES.

c) User Decisions: CogniCrypt asks the user whether the application encrypts large chunks of data. If so, encryption is performed iteratively on fractions of the plaintext instead. Subsequently, it allows the user to decide whether the encryption key should be derived from a password, or created by traditional means of a key generator.

2) Password Storage:

a) Description: transformation of passwords such that they can be securely stored (i.e., hashing and salting).

b) Implementation: implementations of key derivation functions in SunJCE Provider [23] such as PBKDF2.

3) Secure Communication:

a) Description: a cryptographic channel based on the Transport Layer Security (TLS) protocol [8] for securely transporting data from one endpoint to another. The channel ensures confidentiality and integrity of the communicated data as well as authenticity of the communication partners.

b) Implementation: based on the Java TLS implementation in the Java Secure Socket Extension (JSSE) [22].

c) User Decisions: CogniCrypt first asks the user whether they wish to implement the client or the server side of a connection, requesting the corresponding internet-address. For client implementations, if the server is already known, CogniCrypt offers to perform a trial connection to test connectivity and cryptographic parameters. CogniCrypt then allows the user to select the desired security level, providing a safe default option for optimal cryptographic protection. In particular, CogniCrypt disables insecure cryptographic parameters (i.e., cipher suites). This feature is crucial, because TLS has a vast number of parameter choices, and, in principle, allows to configure insecure cipher suites that, for example, omit encryption or enable known attacks like RC4 weaknesses [1].

4) Secure Long-Term Storage:

a) Description: MoPS [28] ensures the integrity and authenticity of documents over long periods of time, since classical protection schemes (e.g., digital signatures) do not

provide everlasting security. MoPS allows users to create customized long-term protection schemes by combining reusable components extracted from other existing solutions, improving performance and gaining flexibility.

b) Implementation: The reference implementation of MoPS by Weinert et al. [28] has a RESTful API for configuring and maintaining file collections on remote systems. Using the API without proper guidance, the user may end up with a configuration that uses outdated cryptographic primitives (e.g., SHA-1), performs poorly due to improper component selection, or relies on inappropriate trust assumptions.

c) User Decisions: CogniCrypt asks the user at most four high-level questions (e.g., "Do you plan to add new files to your collection frequently?"). These questions identify the required features and the trust assumptions the user is willing to make. The Clafer model then translates the user choices into the most suitable component selection based on the recommendations of Weinert et al. [28]. Finally, CogniCrypt generates glue code to configure the MoPS system accordingly and provide methods for securely storing files in the system.

5) Secure Multi-Party Computation:

a) Description: ABY [7] is a framework for mixed-protocol secure two-party computation (STC). It allows two parties to apply a function to their private inputs and reveal nothing but the output of the computation. ABY enables developers to implement STC applications by offering abstractions from the underlying protocols. Furthermore, ABY can securely convert between different protocol types, improving efficiency.

b) Implementation: ABY is written in C/C++ to achieve high efficiency for the underlying primitives (bit operations, symmetric encryption) and has been encapsulated in Java Native Interface (JNI) wrappers to be used by CogniCrypt.

c) User Decisions: CogniCrypt offers the user several STC example applications, e.g., computing the Euclidean Distance between private coordinates. The user can select different properties, depending on the deployment scenario. In the future, we plan to integrate custom applications as well.

B. Implementations of Cryptographic Algorithms

CogniCrypt mainly capitalizes on algorithm implementations from the Java Cryptography Architecture (JCA) [21]. For the first three tasks described above, we have not implemented any cryptographic algorithms ourselves, but merely accessed the existing ones through the JCA APIs. In the future, we would like cryptography experts to contribute new algorithm implementations to CogniCrypt to extend support for even the most novel of cryptographic schemes. The cryptography researchers among the authors have already started integrating an implementation of a novel public-key cryptographic algorithm.

Lindner and Peikert [16] present a new public-key encryption algorithm (*LP11*) based on the learning-with-errors problem. As a lattice-based primitive, it is currently believed to withstand attacks on classical *and* quantum computers, a property typically referred to as *post-quantum security*. For efficiency reasons, we implemented *LP11* in C++ and integrated it into CogniCrypt using JNI. We made three methods of the C++ implementation available for Java: key generation, encryption, and decryption, and implemented the necessary JCA interfaces for encryption and key generation. Not only does this setup allow for an easy integration into CogniCrypt, but it also enables standalone usage of *LP11*.

Unfortunately, the interfaces provided by JCA do not completely fit the properties of post-quantum encryption schemes. In particular, the interface provides two methods for key-pair generation: one bootstrapped with a key size, the other one allowing for more parameters to be included in the key generation. As one key size is not sufficient for *LP11*, our implementation only supports the other method. Calling the incorrect method causes CogniCrypt to alert the developer, and throws an `UnsupportedOperationException` at runtime.

VI. Related Work

We are not aware of any integrated tool that combines code generation and static analysis for misuses of Java cryptographic APIs. However, there has been a number of static analysis tools that detect misuses of cryptographic and other security APIs in Java [6, 9–11, 20, 25]. Unlike CogniCrypt, these tools do not provide any IDE integration and have hard-coded checks. Additionally, CogniCrypt enables cryptography experts, who may not be experts in static analysis, to define new rules more easily through TS4J.

CogniCrypt generates task-based usage examples for Java Crypto APIs. Although similar tools exist [5, 12, 13, 17], they rely on the mining of syntactically correct usages of the respective APIs, which is not a viable approach for cryptographic APIs for two reasons. First, many insecure usages of cryptographic APIs may still result in syntactically correct programs. Second, it appears that most usage examples of cryptographic APIs in the wild are insecure [6, 9, 14], causing the mining of such usages to be a difficult endeavour.

VII. Conclusion

Cryptography can help secure sensitive data, but only if applications use cryptographic components securely. We have presented CogniCrypt, an Eclipse plugin that enables developers to securely integrate such components into their Java projects, especially if they have little experience with cryptography. CogniCrypt smoothly integrates into a developer's workflow to generate secure code for cryptographic tasks and detect misuses of cryptographic APIs in their code.

For now, all tasks have been integrated by the authors. We plan to open-source CogniCrypt toward the end of 2017, and we encourage cryptography experts to integrate their own projects into it. We also plan to conduct a user study to evaluate whether CogniCrypt is capable of improving the security of the average developers' code.

Acknowledgments

This work was funded by the DFG as part of projects P1, S4, S6, E3, and E1 within the CRC 1119 CROSSING as well as by the BMBF and the HMWK within CRISP. We would like to thank Mohammad Hassan Zahraee, Patrick Hill, André Sonntag, and Sneha Reddy for their work on CogniCrypt.

References

[1] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. On the security of RC4 in TLS. In *USENIX Security*, pages 305–320, 2013.

[2] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, and M. Mezini. Towards secure integration of cryptographic software. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2015.

[3] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wąsowski. Clafer: unifying class and feature modeling. *Software and System Modeling*, 15(3):811–845, 2016.

[4] E. Bodden. TS4J: a fluent interface for defining and computing typestate analyses. In *International Workshop on State of the Art in Java Program analysis (SOAP)*, pages 1:1–1:6, 2014.

[5] R. P. L. Buse and W. Weimer. Synthesizing API usage examples. In *International Conference on Software Engineering (ICSE)*, pages 782–792, 2012.

[6] A. Chatzikonstantinou, C. Ntantogian, G. Karopoulos, and C. Xenakis. Evaluation of cryptography usage in android applications. In *International Conference on Bio-inspired Information and Communications Technologies (BIONETICS)*, pages 83–90, 2016.

[7] D. Demmler, T. Schneider, and M. Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[8] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), 2008. Updated by RFCs 5746, 5878, 6176.

[9] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Conference on Computer and Communications Security (CCS)*, pages 73–84, 2013.

[10] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: an Analysis of Android SSL (In)security. In *Conference on Computer and Communications Security (CCS)*, pages 50–61, 2012.

[11] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang. Vetting SSL usage in applications with SSLINT. In *IEEE Symposium on Security and Privacy*, pages 519–534, 2015.

[12] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *International Conference on Software Engineering (ICSE)*, pages 664–675, 2014.

[13] J. Kim, S. Lee, S. Hwang, and S. Kim. Towards an intelligent code search engine. In *Conference on Artificial Intelligence (AAAI)*, 2010.

[14] D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *ACM Asia-Pacific Workshop on Systems (APSys)*, pages 7:1–7:7, 2014.

[15] J. Liang, M. Antkiewicz, A. Murashkin, and J. Ross. Choco solver - A Backend for Clafer using the Choco4 solver, 2016. URL https://github.com/gsdlab/chocosolver.

[16] R. Lindner and C. Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Cryptographers' Track at the RSA Conference (CT-RSA)*, pages 319–339, 2011.

[17] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, and A. Marcus. How can I use this method? In *International Conference on Software Engineering (ICSE)*, pages 880–890, 2015.

[18] S. Nadi and S. Krüger. Variability modeling of cryptographic components: Clafer experience report. In *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 105–112, 2016.

[19] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. Jumping through hoops: why do Java developers struggle with cryptography APIs? In *International Conference on Software Engineering (ICSE)*, pages 935–946, 2016.

[20] L. Onwuzurike and E. D. Cristofaro. Danger is my middle name: experimenting with SSL vulnerabilities in android apps. In *ACM Conf. on Security & Privacy in Wireless and Mobile Networks (WiSec)*, pages 15:1–15:6, 2015.

[21] Oracle. Java Cryptography Architecture (JCA), 2016. URL https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html.

[22] Oracle. Java Secure Socket Extension (JSSE) Reference Guide, 2017. URL https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html.

[23] Oracle. Cipher Implementations in SunJCE Provider, 2017. URL https://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html#SunJCEProvider.

[24] Saxonica. The SAXON XSLT and XQuery Processor, 2016. URL http://saxon.sourceforge.net.

[25] S. Shao, G. Dong, T. Guo, T. Yang, and C. Shi. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *IEEE International Conference on Dependable, Autonomic andSecure Computing (DASC)*, pages 75–80, 2014.

[26] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)*, 12(1):157–171, 1986.

[27] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction (CC)*, pages 18–34, 2000.

[28] C. Weinert, D. Demirel, M. Vigil, M. Geihs, and J. Buchmann. MoPS: A Modular Protection Scheme for Long-Term Storage. In *Asia Conference on Computer and Communications Security (ASIACCS)*, pages 436–448, 2017.