CogniCrypt$_{GEN}$

Generating Code for the Secure Usage of Crypto APIs

Stefan Krüger
Paderborn University
Germany
stefan.krueger@uni-paderborn.de

Karim Ali
University of Alberta
Canada
karim.ali@ualberta.ca

Eric Bodden
Paderborn University & Fraunhofer IEM
Germany
eric.bodden@uni-paderborn.de

Abstract

Many software applications are insecure because they misuse cryptographic APIs. Prior attempts to address misuses focused on detecting them after the fact. However, avoiding such misuses in the first place would significantly reduce development cost.

In this paper, we present CogniCrypt$_{GEN}$, a code generator that proactively assists developers in using Java crypto APIs correctly. CogniCrypt$_{GEN}$ accepts as input a code template and API-usage rules defined in the specification language CrySL. The code templates in CogniCrypt$_{GEN}$ are minimal, only comprising simple glue code. All security-sensitive code is generated fully automatically from the CrySL rules that the templates merely refer to. That way, generated code is provably correct and secure with respect to the CrySL definitions. CogniCrypt$_{GEN}$ supports the implementation of the most common cryptographic use cases, ranging from password-based encryption to digital signatures.

We have empirically evaluated CogniCrypt$_{GEN}$ from the perspectives of both crypto-API developers and application developers. Our results show that CogniCrypt$_{GEN}$ is fast enough to be used during development. Compared to a state-of-the-art template-based solution, implementing use cases with CogniCrypt$_{GEN}$ requires only a fourth of development effort, without any additional language skills. Real-world developers see CogniCrypt$_{GEN}$ as significantly simpler to use than the same template-based solution.

CCS Concepts • **Software and its engineering → Software development techniques**; • **Security and privacy → Software security engineering**; Cryptography.

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377905

Keywords Cryptographic Misuse, Code Generation, Security Specifications, Code Templates

ACM Reference Format:
Stefan Krüger, Karim Ali, and Eric Bodden. 2020. CogniCrypt$_{GEN}$: Generating Code for the Secure Usage of Crypto APIs. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3368826.3377905

1 Introduction

Previous studies have shown that 88%–95% of Android apps misuse APIs of cryptographic libraries [6, 7, 22, 40]. Such misuses may result in dire consequences when badly implemented cryptography is easily broken considering cryptography's role in securing sensitive information in almost all digital devices and transactions [2, 13, 23, 38].

In response to these findings, a number of program-analysis tools [6, 7, 22, 40] and program-repair tools [24, 41] have attempted to combat the misuse of crypto APIs by analyzing a given target program for the misuses and also trying to fix them afterwards. While those attempts are a step forward, such detection and mitigation techniques come at a cost: developers first must integrate the API insecurely to then—hopefully, at some point—learn how to fix the integration, a request that seems questionable when taking into account the results of a study by Nadi et al. [29]. The authors of that study surveyed developers about their experience with crypto APIs. Many of them struggle at least occasionally with bad design (81%) and the knowledge that is required to engage with them (35%). These main issues cannot meaningfully be addressed solely by code-analysis or code-repair tools. While a full re-design of the APIs might provide a more long-term solution, such a step cannot be enforced by the APIs' users, which means that they will still have to struggle with current API designs. To this end, participants in the survey by Nadi et al. [29] also requested tools that would provide secure implementations of common cryptographic programming tasks.

To address these shortcomings, we present CogniCrypt$_{GEN}$, a code generator for secure integrations of crypto APIs. The tool operates on a Java project into which it generates code, and accepts as input a template with interface and glue

code, as well as usage rules in the API-specification language CrySL [22]. CrySL enables domain experts to specify how to use a cryptographic API correctly and securely. Our design of CogniCrypt$_{GEN}$ simplifies the used code templates, causing the vast majority of the code to be generated fully automatically from CrySL definitions. This code is provably correct and secure (assuming a correct and secure specification of CrySL rules by the domain experts). In addition, as CogniCrypt$_{GEN}$ generates code from type- and syntax-checked specifications, the generated code does not contain syntax errors or produce type errors either.

Given a template that refers to a number of CrySL specifications, through a series of model transformations, CogniCrypt$_{GEN}$ translates these specifications into Java code and generates the complete implementation into the provided Java project. We have integrated CogniCrypt$_{GEN}$ into the Eclipse-based crypto assistant CogniCrypt [21], in which it replaces the existing code-generation solution, hereafter referred to as CogniCrypt$_{OLD\text{-}GEN}$. CogniCrypt$_{OLD\text{-}GEN}$ has the drawback of using XSL templates that are disconnected from any CrySL specifications, which frequently lead to inconsistencies between those artefacts. With our integration of CogniCrypt$_{GEN}$, providers of crypto APIs instead need to only develop and maintain a small and simple set of Java code templates and the corresponding set of CrySL rules. The CrySL rules also grant other kinds of tool support from CogniCrypt: CogniCrypt$_{GEN}$ complements the existing tool CogniCrypt$_{SAST}$, which uses the very same CrySL specifications to find misuses in existing application code using static analysis. The many existing CrySL specifications for CogniCrypt$_{SAST}$ can be reused for CogniCrypt$_{GEN}$.

We evaluate CogniCrypt$_{GEN}$ along four different dimensions. First, to determine its expressiveness, we implement templates for eleven common cryptographic use cases in CogniCrypt$_{GEN}$. Second, we measure runtime performance and memory consumption of CogniCrypt$_{GEN}$ on all those use cases. We then compare CogniCrypt$_{GEN}$ to the existing XSL-based code-generation solution CogniCrypt$_{OLD\text{-}GEN}$. In this context, we compare the effort to create and maintain the artefacts for the eight use cases that CogniCrypt$_{OLD\text{-}GEN}$ supports with our own implementation of the same use cases in CogniCrypt$_{GEN}$. Lastly, we investigate the usability of CogniCrypt$_{GEN}$ for a crypto-API developer, i.e., a domain expert who may integrate new use cases or modify existing ones. To this end, we conduct a user study with 16 participants and ask them to perform a series of modifications to artefacts of existing use-case implementations. In conclusion, we manage to implement all common use cases we find in CogniCrypt$_{GEN}$. CogniCrypt$_{GEN}$ further outperforms CogniCrypt$_{OLD\text{-}GEN}$ in terms of usability and maintainability, while showing negligible memory overhead and fast performance results of below ten seconds.

```
1  public Key generateKey(String pwd) {
2    byte[] salt = {15, -12, 94, 0, 12, 3, -65, 73, -1,
         -84, -35};
3    PBEKeySpec spec = new PBEKeySpec(
         pwd.toCharArray(), salt, 100000, 256);
4
5    SecretKeyFactory skf = SecretKeyFactory.
         getInstance("PBKDF2WithHmacSHA256");
6    byte[] keyMaterial =
         skf.generateSecret(spec).getEncoded();
7
8    SecretKeySpec cipherKey = new
         SecretKeySpec(keyMaterial, "AES");
9    return cipherKey;
10 }
```

Figure 1. An example illustrating the *incorrect* implementation of a password-based encryption (PBE) in Java.

2 Programming with Cryptography in Java

We now briefly describe how cryptography is implemented in Java. We will also introduce the elements of the specification language CrySL [22] that CogniCrypt$_{GEN}$ uses.

2.1 Java Cryptography

The most commonly used cryptographic library is the Java Cryptography Architecture (JCA) [35]. The JCA serves both as a set of APIs as well as a default implementation for these APIs that is also shipped with the JDK. The underlying infrastructure is designed such that other implementations (i.e., providers) may be easily plugged into the interfaces. In our work, we focus on the default JCA implementation that provides a wide range of cryptographic services, including symmetric and asymmetric encryption, digital signatures, and key management.

For the purpose of password-based data encryption (PBE), the JCA offers the `PBEKeySpec` and `Cipher` APIs. The former is used for the derivation of a cryptographic key from a password, and the latter for the actual encryption. Figure 1 presents a potential use of `PBEKeySpec` that is widespread in real-world applications. While the example avoids some common misuses (e.g., using a low iteration count) [3, 6, 7, 22, 40], it is still insecure for the following non-trivial reasons.

The method `generateKey()` begins with the setup of the encryption key by creating a `PBEKeySpec` object. The constructor of `PBEKeySpec` expects a password, a salt, an iteration count, and a key length. The `PBEKeySpec` object, once created, is passed to a `SecretKeyFactory` that uses the password-based key-derivation algorithm `PBKDF2` to generate key material (Lines 5–6). Based on this key material, Line 8 generates a key for the symmetric cipher AES.

As intuitive as this snippet may look, it contains three misuses that make the code insecure, all are related to `PBEKeySpec`. Three of the four arguments to the constructor (Line 3) have usage constraints, and the example breaks two of them. Let

us first investigate the parameter that it sets correctly: the iteration count. There is no general consensus on what a secure iteration count is, but 100,000 is well above most recommendations [5, 10, 14]. The first misuse relates to the constant salt that is passed to the constructor call. To avoid the possibility of pre-computing rainbow tables [33], salts must be generated through a cryptographically secure random source. The second misuse is for the argument `pwd`. The constructor of `PBEKeySpec` expects the password to be of type `char[]`, and it does so for a good reason. Passwords should not remain accessible in memory any longer than absolutely necessary. Unlike arrays, strings are immutable in Java. Whenever a string is modified, a new string is created, and the old one is kept in memory until garbage-collected. To limit the password's lifetime in memory, developers must favour using `char[]` over `String` for passwords and clear the array after passing it to the constructor. The third misuse is related to the fact that `PBEKeySpec` does not automatically clear the password soon after use. Instead, it expects the developer to call `clearPassword()` after the object has fulfilled its purpose, which the code in Figure 1 does not do.

Although the code contains these security-breaking misuses, it nonetheless runs without throwing exceptions. Not only must developers make sure to use the API in a functionally correct way, they also must consider the code's security properties. This scenario is especially concerning considering that Java does not provide any tool support to detect insecure uses. Many developers reuse code snippets from online resources such as StackOverflow that are frequently functionally correct but insecure [9].

2.2 Specifying Secure Usage of Cryptographic APIs

CrySL is a domain-specific language that enables experts to define how a certain API should be used. CrySL's syntax is close to Java. Krüger et al. [22] provide further information about the design decisions that underly CrySL.

Generally speaking, CrySL specifications follow a whitelisting approach, treating all deviations from a specification as misuses. In CrySL, each rule specifies the correct use of one Java class or interface. A rule may comprise four sections: (1) a calling-sequence pattern, (2) constraints on parameters or combinations thereof, (3) forbidden methods, and (4) constraints on how the specified object may be composed with objects of other types. None of these sections are mandatory.

To illustrate the capabilities of CrySL, let us consider the CrySL rule for `PBEKeySpec` [20, 22], shown in Figure 2. The rule starts off by stating the class it specifies. Subsequently, the **`OBJECTS`** section defines four objects that may be referenced in the subsequent parts of the rule. One such object is the `char[] password`. The **`EVENTS`** and **`ORDER`** sections specify the usage pattern. First, within **`EVENTS`**, one defines all methods that may contribute to a successful use of `PBEKeySpec` as method-event patterns (Lines 20–21). The **`ORDER`** section then defines valid execution sequences for the

```
11 SPEC javax.crypto.spec.PBEKeySpec
12 OBJECTS
13   char[] password;
14   byte[] salt;
15   int iterationCount;
16   int keylength;
17
18 ...
19 EVENTS
20   c1: PBEKeySpec(password, salt,
         iterationCount, keylength);
21   cP: clearPassword();
22
23 ORDER
24   c1,  cP
25
26 CONSTRAINTS
27   iterationCount >= 10000;
28   ...
29
30 REQUIRES
31   randomized[salt];
32 ENSURES
33   speccedKey[this, keylength] after c1;
34 NEGATES
35   speccedKey[this, _];
```

Figure 2. CrySL rule for using the JCA class `javax.crypto.spec.PBEKeySpec`.

```
36 SPEC javax.crypto.SecretKeyFactory
37 OBJECTS
38     int keylength;
39     java.lang.String cipherAlg
40 ...
41 REQUIRES
42   speccedKey[keylength];
43 ENSURES
44   generatedKey[this, cipherAlg];
```

Figure 3. CrySL rule for using the JCA class `javax.crypto.SecretKeyFactory`.

method calls defined this way. CrySL comes with a number of syntax elements that support and simplify the specification process such as placeholders to ignore certain parameters, combining multiple calls in one line, labelling patterns, and aggregating labels. The usage pattern for `PBEKeySpec` is rather simple. It requires a call to the secure constructor labelled `c1` followed by a call to `clearPassword()` to prevent the third misuse in Figure 1. The **`CONSTRAINTS`** section adds constraints over parameter values, i.e., the objects declared in the **`OBJECTS`** section. The `PBEKeySpec` rule requires the `iterationCount` variable to be at least 10,000.

To cover constraints between classes, CrySL offers two keywords that implement a rely/guarantee reasoning. First,

the **ENSURES** section defines predicates that serve as guarantees a class provides if a given use of that class adheres to its CrySL rule. In other words, an object is used securely, i.e., ensures its predicates, if and only if the use under analysis follows the defined method sequence, does not violate any parameters constraints, and avoids calling forbidden methods. The rule PBEKeySpec makes use of another keyword in the **ENSURES** section: through the keyword `after`, it ensures the predicate `speccedKey` only after the call to the constructor `c1`. In addition, the rule defines another section **NEGATES** (Line 35) to denote that the `speccedKey` predicate ought to be invalidated after `cP` is called. This rule may look confusing at first, but makes sense when put in context. The second method `clearPassword()` nullifies the array, as described above. Therefore, any given PBEKeySpec object can only ever fulfil its purpose as a key specification after executing the class's constructor but before executing `clearPassword()`.

While the predicate is ensured, other classes may in turn rely on it using the **REQUIRES** section (Line 42 in Figure 3). For a given use of `SecretKeyFactory` to be correct, the object must adhere to the rule's usage pattern and parameter constraints. Additionally, any key specification flowing into it must also ensure the `speccedKey` predicate, signalling its secure generation. Class compositions are not a rare phenomenon. PBEKeySpec in Figure 2 must rely on the proper use of another class, too. In Line 31, the rule requires the object `salt` to come with the predicate `randomized`. At the time of writing, only `SecureRandom` can grant this predicate.

Overall, the CrySL rule for PBEKeySpec mitigates against all three misuses that we have identified above. Thanks to its largely white-listing approach and the already developed comprehensive CrySL rule set for the JCA [20], we opted to take CrySL as a building block for CogniCrypt$_{GEN}$.

3 Generating Secure Code from CrySL

In this section, we present CogniCrypt$_{GEN}$. First, we discuss the goals and considerations of our design. We then go further into its API, the code templates that interface CogniCrypt$_{GEN}$, and the underlying code-generation algorithm.

3.1 Design Considerations

CogniCrypt$_{GEN}$ targets a very practical problem: the widespread misuse of crypto-APIs. To this end, we put usability at the center of the design effort. Usability first and foremost refers to usability for regular Java developers. To truly help those developers, CogniCrypt$_{GEN}$ must support the most common cryptographic use cases. It must also integrate into the development workflow of a regular developer, which puts limitations on the running time of CogniCrypt$_{GEN}$. On top of these aspects, prioritizing usability, for us, also means to have a usable development and maintenance process of the artefacts in the backend of CogniCrypt$_{GEN}$ (i.e., CrySL rules and code templates). The target audience on

```
45  public SecretKey generateKey(char[] pwd) {
46    byte[] salt = new byte[32];
47    javax.crypto.SecretKey encryptionKey = null;
48
49    CrySLCodeGenerator.getInstance().
50    considerCrySLRule("java.security.SecureRandom").
51    addParameter(salt,"salt").
52    considerCrySLRule("java.security.PBEKeySpec").
53    addParameter(pwd, "password").
54    considerCrySLRule("javax.crypto.SecretKeyFactory").
55    considerCrySLRule("java.security.SecretKey").
56    considerCrySLRule("javax.crypto.SecretKeySpec").
57    addReturnObject(encryptionKey).generate();
58    return encryptionKey;
59  }
```

Figure 4. CogniCrypt$_{GEN}$ template that generates a correct Java implementation for PBE from Figure 2.

the backend of CogniCrypt$_{GEN}$ are crypto experts who have either developed a library that they want to integrate into CogniCrypt$_{GEN}$ or have some experience with one. To keep things simple, we opted for Java code templates as opposed to other template languages that Java crypto experts are not likely familiar with. In addition, CrySL has a Java-like syntax as Krüger et al. [22] have previously argued. While CrySL is an additional artefact that may not be needed by other template engines in particular, or code-generation approaches in general, developing rules in CrySL comes with the additional advantage of gaining other tool support from the CogniCrypt ecosystem. The result of our design is a code generator combining code templates with CrySL rules. Thanks to this setup, CogniCrypt$_{GEN}$, in conclusion, guarantees to generate code code that (1) is free of syntax errors, (2) type-checks in Java, and (3) does not violate CrySL rules.

3.2 Configuring Solutions with Java Code Templates

CogniCrypt$_{GEN}$'s code templates are regular Java classes. They allow crypto experts to (1) include use-case-specific wrapper code, (2) specify CrySL rules that make up the given use case, and (3) pass objects from the wrapper code to CogniCrypt$_{GEN}$. Figure 4 shows the code template for implementing the PBE example from Figure 2 *correctly and securely*. Figure 5 shows how the generated code uses this template. The first line of the template defines a `salt`. This explicit definition is necessary because the involved APIs require a byte array, but do not create one themselves. The line after defines a cryptographic key called `encryptionKey`, which the generated code uses to store the generated key (Line 74).

Starting at Line 49, the template calls CrySL rules and instantiates their parameters using a fluent API [11]. The call to `getInstance()` instantiates the code generator. Line 50 includes the class `java.security.SecureRandom` into the code generation through a call to `considerCrySLRule()`.

```
61 public class TemplateClass {
62  public SecretKey generateKey(char[] pwd) {
63   byte[] salt = new byte[32];
64   SecretKey encryptionKey = null;
65
66   SecureRandom secureRandom =
         SecureRandom.getInstance("SHA1PRNG");
67   secureRandom.nextBytes(salt);
68   PBEKeySpec pBEKeySpec = new PBEKeySpec(pwd, salt,
         27799, 128);
69
70   SecretKeyFactory secretKeyFactory =
         SecretKeyFactory.getInstance
71       ("PBEWithHmacSHA512AndAES_128");
72   SecretKey secretKey =
         secretKeyFactory.generateSecret(pBEKeySpec);
73   byte[] keyMaterial = secretKey.getEncoded();
74   encryptionKey = new SecretKeySpec(keyMaterial,
         "AES");
75
76   pBEKeySpec.clearPassword();
77   return encryptionKey;
78  }
79 }
80 public class OutputClass {
81   public void templateUsage(char[] pwd) {
82     TemplateClass tc = new TemplateClass();
83     SecretKey key = tc.generateKey(pwd);
84   }
85 }
```

Figure 5. Code Generated by CogniCrypt$_{GEN}$ using the template in Figure 4.

In the next line, the call to `addParameter()` associates the byte array `salt` in the template with the variable `salt` in the CrySL rule for `SecureRandom`. Lines 50–51 properly randomize `salt` before using it during key generation. Lines 52–56 then create a `java.security.PBEKeySpec` key and transform it into a `javax.crypto.SecretKeySpec` key. Finally, the call to `addReturnObject()` assigns `encryptionKey` the role of a return object. During the generation, the return value of the constructor of `javax.crypto.SecretKeySpec` is stored in `encryptionKey`. CogniCrypt$_{GEN}$ selects the constructor because it is the last method of that class that needs to be called according to the CrySL rule. As a result, the key based on the password is assigned to `encryptionKey`. Line 58 returns the key as part of the boilerplate code.

3.3 Generating Secure Code from Templates

Figure 6 shows the workflow of CogniCrypt$_{GEN}$. In the following, we describe how it works and will refer to the individual steps using their corresponding numbers in the figure. For each call to the code-generator API in a template, CogniCrypt$_{GEN}$ first collects all rules and their parameters from an API call chain ①. The chain in Line 49 in method `generateKey()` requires rules for `java.security.SecureRandom`, `java.security.PBEKeySpec`, `javax. crypto.SecretKeyFactory`, and `javax.crypto.SecretKeySpec`. In addition, the CrySL rules for `java.security.SecureRandom` and `java.code.PBEKeySpec` get attached the objects `salt` and `pwd` to their in-rule variables `salt` and `password`, respectively. Lastly, the rule for `SecretKeySpec` yields the object `encryptionKey` as a return object.

CogniCrypt$_{GEN}$ then iterates through the rules to assemble a list of predicates that link rules to one another ②. These links form a path that CogniCrypt$_{GEN}$ uses to select appropriate method sequences for a given class ③. If two classes are connected through a predicate, CogniCrypt$_{GEN}$ may, for the class that should *ensure* the predicate, only select method sequences that eventually grant this predicate. Similarly, for the class that *requires* the predicate, CogniCrypt$_{GEN}$ picks method sequences that make use of the predicate. For the PBE example in Figure 4, `PBEKeySpec` can generate the predicate `speccedKey` on itself. `SecretKeyFactory`, in turn, requires this predicate on the key specification object that it uses to generate a key (Line 42 in Figure 3). Hence, CogniCrypt$_{GEN}$ connects both rules using the predicate `speccedKey` on the `PBEKeySpec` object. If CogniCrypt$_{GEN}$ were unable to establish a path between `PBEKeySpec` and `SecretKeyFactory`, it would not have taken the former into account when generating code for the latter.

Next, CogniCrypt$_{GEN}$ iterates through all rules again to assemble the code, which includes (1) generating method calls ③ for all involved classes and (2) finding appropriate values for their parameters ④. For each CrySL rule, CogniCrypt$_{GEN}$ first compiles a list of correct paths of method calls according to the specified calling-sequence pattern ③. To this end, CogniCrypt$_{GEN}$ translates a rule's pattern into a finite state machine. The tool then classifies any path of method calls that leads to an acceptable state in the state machine as correct. When assembling such paths, CogniCrypt$_{GEN}$ has to deal with methods that, according to the state machine, may be called multiple times. CogniCrypt$_{GEN}$ translates such methods into two different paths: one where the method is not called and one where it is. CogniCrypt$_{GEN}$ does not currently support repeated calls. However, in our experiments with the JCA, this lack of support has not proven to be a problem. In scenarios where more than one correct path is found, CogniCrypt$_{GEN}$ applies a set of filters to reduce the number of sequences. Paths that do not include objects required by the code template through calls to `addParameter()` cannot implement the use case and are, therefore, eliminated. For `PBEKeySpec`, its CrySL rule in Figure 2 prescribes one specific constructor (`c1`) and the method `clearPassword()` to be called in that order. Therefore, for this class, CogniCrypt$_{GEN}$ finds only this one possible path to an accepting state. Similarly, CogniCrypt$_{GEN}$ discards paths that may lead to different predicates than the ones associated with it. In the case of `PBEKeySpec`, the only possible

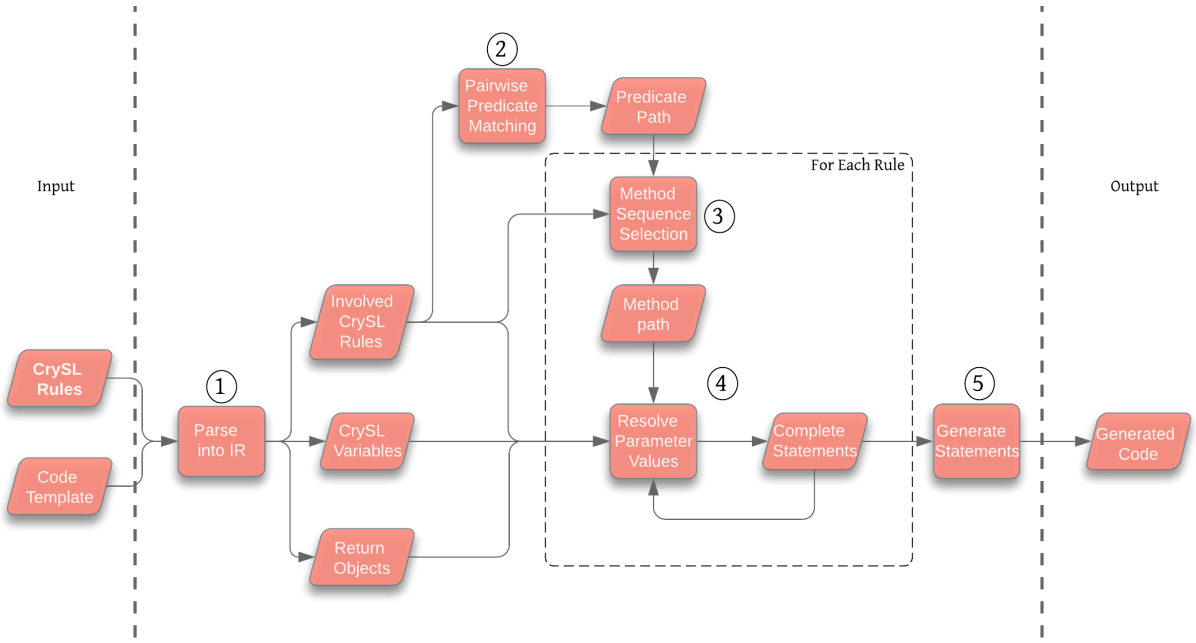

Figure 6. The General Workflow of CogniCrypt$_{GEN}$.

path grants the correct predicate `speccedKey`. Consequently, CogniCrypt$_{GEN}$ does not remove this path from the list of possible paths to take. Instead, CogniCrypt$_{GEN}$ generates the call to the `PBEKeySpec` constructor in Line 68 of Figure 5.

The second call, the call to `clearPassword()`, CogniCrypt$_{GEN}$ appends to the block of API statements in Line 76 before the return statement because the method invalidates the `speccedKey` predicate of the object. Instead of generating calls to such invalidating methods directly, CogniCrypt$_{GEN}$ collects them and generates them at the end of the method.

For each method call on the remaining paths, CogniCrypt$_{GEN}$ applies several heuristics to resolve possible parameter values ④. First, it attempts to match parameters required by the template through calls to `addParameter()` to a given parameter in a method call. Two objects match when the CrySL variable mapped to an object in the `addParameter()` call is the one used in the method call. At Line 68 of Figure 5, CogniCrypt$_{GEN}$ matches `password`, the first parameter in the constructor `c1` of `PBEKeySpec`, to the argument `pwd` of the method `generateKey()`. It does so because the call to `addParameter()` maps `pwd` to the CrySL variable `password` in `PBEKeySpec` (Line 53). In case of no match, CogniCrypt$_{GEN}$ further attempts to match a parameter to objects in the generated code that have received a matching predicate. For the PBE example, CogniCrypt$_{GEN}$ generates code for `SecureRandom` that ensures the predicate `randomized` for `salt`. CogniCrypt$_{GEN}$ then matches the second parameter in the call to the `PBEKeySpec` constructor to this `salt` object because it has the same type and requires the same predicate. CogniCrypt$_{GEN}$ conducts one further step to resolve remaining method parameters that may not be matched to any existing objects from the code template or the generated code. For those, it queries constraints from the respective CrySL rule and fetches secure values from the first appropriate constraint that it finds. For value constraints of the form `var in {Literal1, ..., LiteralN}`, it selects the first option `Literal1`. This type of constraint usually comes into play for algorithms and key sizes (e.g., in `KeyGenerator` or `Cipher` [20]). From a security perspective, since all values in a CrySL rule ought to be correct, it does not matter which value CogniCrypt$_{GEN}$ chooses. Another type of constraints becomes relevant for the third parameter of the `PBEKeySpec` constructor. According to the rule in Figure 2, the iteration count must be $\geq$ 10,000. CogniCrypt$_{GEN}$ generates the closest value that satisfies this constraint, which is 10,000 (Line 68 in Figure 5).

For cases in which this attempt to resolve the parameter fails as well, we decided to prioritize compilability of the generated source code over completeness. That is, CogniCrypt$_{GEN}$ adds the unresolvable parameter to the wrapper method that the call belongs to. During the development of our code templates, this feature has proven useful for debugging. We have first specified which CrySL rules should be included in the generation and after running CogniCrypt$_{GEN}$, the generated code showed which parameters needed additional specification. We believe this feature will also help crypto experts in similar situations. For the final code template, however, this step is meant as a fallback solution because it changes the wrapper method as defined in the code template and de-facto complicates the use of the method. In practice, CogniCrypt$_{GEN}$ did not have to take this final step for any of the use cases we have implemented.

If, at the end of this process, CogniCrypt$_{GEN}$ needs to choose between multiple method paths with fully-resolvable parameters, it selects the shortest one. That is, CogniCrypt$_{GEN}$ opts for the method path with the fewest method calls as well as the smallest number of parameters. When all calls are selected and all parameter objects have been assigned values, CogniCrypt$_{GEN}$ generates the produced code into the target program ⑤. This process is then repeated for all calls to CogniCrypt$_{GEN}$'s fluent API within a given template. Once CogniCrypt$_{GEN}$ has processed a template, it also generates a method that showcases the usage of the generated code. To this end, it creates a new class and method, in which it first instantiates an object of the template class. For our running example, Line 81 in Figure 5 marks the first statement of the corresponding method. In that method, CogniCrypt$_{GEN}$ iterates through all methods of the template class that contains calls to the fluent API and generates calls for them. For `generateKey()` in the template, the generated call is at Line 83. Other methods are assumed to be internal helper methods. CogniCrypt$_{GEN}$ further attempts to match parameters by type-matching a given parameter to return values of previous calls. To ensure compilability, CogniCrypt$_{GEN}$ pushes up parameters where no matching is possible, e.g., `pwd` of `generateKey()`, to become parameters of method `templateUsage()`. For `pwd`, this is indeed the correct behaviour, as the password should be an input rather than a hard-coded value. In conclusion, we view this method as useful for developers, because they do not need to engage with the generated code, but only with this summary method. We drew inspiration for this feature from CogniCrypt's previous code-generator CogniCrypt$_{OLD\text{-}GEN}$, in which such a method is hard-coded into the tool's templates.

4 Implementation Details

We developed CogniCrypt$_{GEN}$ on top of the existing infrastructure for CogniCrypt. We had to first modify CrySL in one aspect. For encryption, the JCA offers one API for both asymmetric and symmetric encryption: `Cipher`. So far, the rule for `Cipher` included one long list of secure algorithms indiscriminately of whether they are performing the former or the latter. For the purpose of program analysis, such a distinction does not necessarily need to be made (although it improves precision to do so). However, implementing use cases involving hybrid encryption (see Section 5) requires differentiating between them. For that purpose, we introduced a new built-in predicate `instanceof(cryslVariable, javaType)`. By means of this predicate, the CrySL rule for `Cipher` now only allows symmetric-encryption algorithms when a key used for encryption is of type `SecretKey` or subtypes(i.e., `instanceof(key, java.security.SecretKey)`. Asymmetric encryption algorithms may only be used when the key is either a private or a public key, indicating that the `Cipher` object implements an asymmetric encryption.

We have further re-used the CrySL parser that Krüger et al. [22] had developed for CogniCrypt$_{SAST}$. However, we needed to modify the existing JCA rule set in the following ways. For some rules (e.g., `Signature` and `KeyGenerator`), we changed the position of arguments in the constraints `var in {Literal1, ..., LiteralN}` to better reflect the preferences in algorithm selection that CogniCrypt$_{GEN}$ should follow. We have also added a new parameter to some predicates (e.g., `Signature`), where the first parameter was not the return value of a cryptographic operation (e.g., the boolean return value of `Signature.verify()`). In all respective cases, CogniCrypt$_{GEN}$ requires the return value to store it in the correct variable as assigned in the template through a call to `addReturnObject()`.

We implemented our own custom solution to template parsing, the traversal of CrySL rules, and code modification. This solution builds on top of the Eclipse Java Development Toolkit (JDT). To parse the templates and apply changes to them in the target project, we have followed a visitor pattern using the JDT's abstract-syntax-tree (AST) APIs.

Prior to our work, CogniCrypt used a CrySL-independent code-generation tool, CogniCrypt$_{OLD\text{-}GEN}$ [21]. Using XSL templates, use-case specific code and points of variability are defined. An algorithm model in the variability-modelling language Clafer [18] supplies correct values (i.e., algorithms) based on user input [28]. We further compare CogniCrypt$_{GEN}$ to CogniCrypt$_{OLD\text{-}GEN}$ in Sections 5 and 6. To improve CogniCrypt, we have replaced CogniCrypt$_{OLD\text{-}GEN}$ with CogniCrypt$_{GEN}$ which implements the eight JCA use cases that CogniCrypt supports: password-based encryption for the data types (1) file, (2) string, and (3) byte array, hybrid encryption for the data types (4) file, (5) string, and (6) byte array, (7) digital signing, and (8) secure password storage.

5 Evaluation

To evaluate CogniCrypt$_{GEN}$, we aim to answer the following research questions:

- RQ1: Can CogniCrypt$_{GEN}$ implement common cryptographic use cases?
- RQ2: Does CogniCrypt$_{GEN}$ produce code quickly enough to be used in everyday software development?
- RQ3: What is CogniCrypt$_{GEN}$'s memory consumption?
- RQ4: What is the effort to create and maintain the artefacts for CogniCrypt$_{GEN}$ to implement common cryptographic use cases?
- RQ5: Do contributors to CogniCrypt$_{GEN}$ perceive a usability gain compared to a state-of-the-art solution using XSL?

With the first three research questions, we aim to determine whether CogniCrypt$_{GEN}$ may actually support developers. If CogniCrypt$_{GEN}$ is incapable of implementing the most

common cryptographic use cases (RQ1), it cannot meaningfully reduce cryptographic misuse. Similarly, if its memory consumption and runtime exceed the average capabilities of workstations (RQ2 + RQ3), application developers will not use it. RQ4 and RQ5 then focus on crypto developers who wish to implement use cases for their own APIs. If CogniCrypt$_{GEN}$ requires too much effort (RQ4) or developers do not find it intuitive to use (RQ5), it is unlikely that a crypto developer will integrate their APIs with CogniCrypt$_{GEN}$, even if they they get access to other tool support in CogniCrypt.

5.1 Implementation of Common Use Cases (RQ1)

Setup To answer RQ1, we have first gathered common cryptographic use cases from multiple sources. The first author of this paper then attempted to implement them with CogniCrypt$_{GEN}$. To check the validity of the generated code with respect to compilability and security, we have further run the Java compiler and CogniCrypt$_{SAST}$ on them.

Results We first collected eight of eleven cryptographic use cases that CogniCrypt$_{OLD\text{-}GEN}$ [21] supports and that use the JCA. We discard the three remaining ones because no CrySL rules exist for them. In their study, Nadi et al. [29] compiled a list of common usage scenarios by (1) analyzing the implemented use cases of 100 randomly selected GitHub projects that implement Java cryptography and (2) asking participants for cryptographic programming tasks that they commonly have to implement. We have also collected the responses often found in projects and popular with participants. Lastly, Mindermann and Wagner [27] have collected an online repository that aims at providing secure implementations for common cryptographic use cases. We have included those use cases into our list as well. Table 1 shows all use cases that CogniCrypt$_{GEN}$ supports as well as their respective sources.

We have successfully implemented all eleven use cases. The implementations of use cases 1–3 are virtually the same in CogniCrypt$_{GEN}$, because they involve the same classes, which leads to having the exact same calls to CogniCrypt$_{GEN}$'s fluent API . Only the wrapper code around the fluent-API calls changes depending on the data type (i.e., File, String, or Byte Array) that is encrypted. The same is true for use cases 5–7 that all deal with hybrid encryption but on different data types. None of the generated code snippets cause compiler errors or true misuses identified by CogniCrypt$_{SAST}$.

5.2 Performance (RQ2 and RQ3)

Setup To answer RQ2, we measure the average running time for each use case in Table 1. To compute the average, we ran each use case ten times, and collect the measurements using `java.lang.System.currentTimeMillis()`.

To answer RQ3, we ran CogniCrypt$_{GEN}$ for each task again. We capture the memory consumption of the Eclipse process through the system memory monitor both before and during the CogniCrypt$_{GEN}$'s run. We then subtract the before value from the highest value during the run. Please note, in pre-experiments, we ran several use cases multiple times, but found the fluctuation in memory usage as negligible (within half a megabyte). We hence decided to take the memory consumption from only a single run.

We ran the experiments on a Windows 10 machine with four CPUs running at 2.6GHz and 16 GB of RAM. We executed all runs on an Eclipse 2019-06 for RCP and RAP developers using Java 8.

Results We list the results for RQ2 and RQ3 in the last two columns of Table 1. CogniCrypt$_{GEN}$ takes between 6.6 and 8.1 seconds. Therefore, all runs are well below ten seconds, making CogniCrypt$_{GEN}$ easily integratble into a developer's programming workflow.

In terms of memory consumption, CogniCrypt$_{GEN}$ consumes between 2.5 and 66.6 MB on top of the regular Eclipse process. During our experiments, the latter oscillated between 900 MB and 1.2 GB of RAM. We conclude that CogniCrypt$_{GEN}$'s memory overhead is negligible.

5.3 Artefact Creation and Maintenance (RQ4)

Setup To approximate the effort of rule creation and maintenance, we compare the artefacts needed to implement the eight cryptographic use cases in CogniCrypt$_{OLD\text{-}GEN}$ to their implementations in CogniCrypt$_{GEN}$. In particular, we compare the total number of lines of code a crypto expert would have to write as well as the language skills required by a developer to implement a use case using both code generators.

We have only investigated artefacts that are specific to the respective code generator. That is, for CogniCrypt$_{OLD\text{-}GEN}$, we looked at Clafer model and XSL code templates. For CogniCrypt$_{GEN}$, on the other hand, we only investigated the code templates, not the involved CrySL rules. CrySL rules, in our case, had been developed independently of this work and have only been changed marginally by us. In general, they are not CogniCrypt$_{GEN}$ specific, but are instead developed to receive general support for an API by CogniCrypt.

Results Table 2 shows the sizes of the different artefacts for the eight use cases that CogniCrypt$_{OLD\text{-}GEN}$ supports. Overall, a developer needs to write at least 111 lines of XSL code and 43 lines of Clafer model to support any of those use cases in CogniCrypt$_{OLD\text{-}GEN}$. On average, each use case implements 136 lines of code in XSL and 91 lines in Clafer. In contrast, a developers needs to write an average of only 60 lines of Java code in CogniCrypt$_{GEN}$ to implement those use cases. Writing less code has two advantages. First, artefacts maintainers need to only keep track of around 25% of the lines of code. Second, crypto experts who have implemented a library in Java do not need to learn extra languages (i.e., Clafer and XSL) to implement their use cases in CogniCrypt$_{GEN}$. Instead, they may define their security code entirely in Java, a language they must be familiar with to implement their

Table 1. Common Cryptographic Use Cases

#	Use Case	Source	Runtime in CC$_{GEN}$ (in s)	Memory Consumption in CC$_{GEN}$ (in MB)
1	PBE on Files	[21]	7.0	14.1
2	PBE on Strings	[21], [27]	6.7	13.5
3	PBE on Byte-Arrays	[21]	7.1	66.6
4	Symmetric-Key Encryption	[27], [29]	6.8	6.0
5	Hybrid File Encryption	[21]	6.7	2.5
6	Hybrid String Encryption	[21]	6.6	4.2
7	Hybrid Byte-Array Encryption	[21]	6.9	56.7
8	Asymmetric String Encryption	[27]	6.8	34.1
9	Secure User-Password Storage	[21], [27]	8.1	22.7
10	Digital Signing of Strings	[21], [27], [29]	7.5	7.1
11	Hashing of Strings	[27]	6.7	14.2

Table 2. Comparing the required lines of code (LOC) to implement the use cases of CogniCrypt$_{OLD\text{-}GEN}$ in both CogniCrypt$_{OLD\text{-}GEN}$ and CogniCrypt$_{GEN}$.

	LOC in CogniCrypt$_{OLD\text{-}GEN}$		LOC in CogniCrypt$_{GEN}$
#	XSL	Clafer	Java
1	140	117	57
2	138	117	57
3	111	117	51
5	158	90	74
6	156	90	74
7	129	90	68
9	139	67	55
10	115	43	40

cryptographic library in the same language. When defining code templates in an IDE such as Eclipse, the crypto experts receive the more advanced development support for Java (e.g., type checking and auto-compiling) compared to what editors or IDEs provide for XSL or Clafer. Those advantages carry over to scenarios where experts may use CrySL in domains other than cryptography.

5.4 Usability (RQ5)

Setup To answer RQ5, we conducted a small-scale user study with 16 participants. We recruited the participants among graduate students at our local university. Each participant is given two tasks, both of which we based on common cryptographic use cases in Table 1, one with CogniCrypt$_{GEN}$ and one with CogniCrypt$_{OLD\text{-}GEN}$.

We choose to compare against CogniCrypt$_{OLD\text{-}GEN}$ for two reasons. First, it is the tool with closest aim as CogniCrypt$_{GEN}$. Second, XSL, as an approach to template-based code generation, provides the ideal setting to compare against. Templates are defined in an extra language with extra features, but still providing a way to write Java code directly. Domain experts might find the additional layer of abstraction this extra language provides useful because it produces a clear cut between code template and generated code. We, however, assume this to not be the case, at least for cryptographers, because, from our prior experience working with cryptographers, we can report that they often do not know any template languages. If participants of our study, who by and large also lack experience with template languages, nonetheless preferred CogniCrypt$_{OLD\text{-}GEN}$ despite the extra language, we would expect domain experts also favouring the old code generator rather than the new one.

Consequently, we designed the tasks such that they rely heavily on modifying code templates, instead of Clafer and CrySL rules. Task 1, based on use case ten in Table 1, asks participants to (1) change a solution that hashes strings to one that hashes files and (2) fix the name of the chosen algorithm that the code generator produces. Task 2, based on use case four in Table 1, asks participants to (1) add proper randomization of an initialization vector for symmetric encryption and (2) prohibit the code generator from using an outdated algorithm. To avoid learning and other carry-over effects, we follow a latin-square approach [12] when randomly assigning tasks and code generators to participants. We give participants 30 minutes to complete each task. Before participants start solving the tasks, the respective instructor gives a 25-minute introduction to both code-generation tools performing the same two modifications on use case eleven in Table 1. After participants have completed their work on the tasks, we ask them to fill a short survey about the perceived usability of the two approaches. We also conducted 5-minute interviews with participants after they have completed the tasks and the survey.

To determine the effectiveness of both code generators, we measure the time that participants need to complete each task. To measure preference for one approach over the other,

we employ the System Usability Scale (SUS) [4] and Net Promoter Score (NPS) [39]. The former determines usability, while the latter rather targets user satisfaction with a system. Both scales transform answers to a questionnaire given by users of a system into a single number. In SUS, a system receives a score between 0 and 100 such that higher values indicate higher usability. Tools that surpass 68 are seen as usable [4]. NPS may range from -100 to +100. Systems that score below 0 are considered unsatisfactory, while results above 50 are viewed as having excellent satisfaction [39]. By means of the post-study survey, we collected more direct feedback and suggestions for improvements for both CogniCrypt$_{GEN}$ and CogniCrypt$_{OLD\text{-}GEN}$.

Results All 16 participants successfully completed both tasks in the given time window. On average, the encryption task was completed 38% slower with CogniCrypt$_{GEN}$ than with CogniCrypt$_{OLD\text{-}GEN}$. In contrast, participants were 63.2% faster to complete the hashing task using CogniCrypt$_{GEN}$ than CogniCrypt$_{OLD\text{-}GEN}$. Investigating the overall completion times, we found no statistical significance with a Wilcoxon signed-rank test for paired data ($p > 0.05$). Initially, the mixed results came as a surprise to us. However, after evaluating the post-study interviews, we are able to attribute them to the steep learning curve for CogniCrypt$_{GEN}$. Seven out of 16 participants mentioned this issue unprompted, all of whom explained that they had not remembered all details from the introduction for either tool. However, since CogniCrypt$_{OLD\text{-}GEN}$ requires more hard-coding, participants managed to get faster to implementing the requested changes. For CogniCrypt$_{GEN}$ on the other hand, they had to re-read the respective existing code to remember the underlying concepts. All seven participants mentioned that a written example-driven documentation that covered what was discussed in the introduction would likely solve this issue.

In terms of usability, CogniCrypt$_{GEN}$ fares significantly better (SUS: 76.3 and NPS: 56.3) compared to CogniCrypt$_{OLD\text{-}GEN}$ (SUS: 50.8 and NS: -43.7). Applying a Wilcoxon signed-rank test, we found the differences between CogniCrypt$_{GEN}$ and CogniCrypt$_{OLD\text{-}GEN}$ in both SUS and NPS to be statistically significant ($p = 0.005$). Overall, participants appreciated the purpose and design of CogniCrypt$_{GEN}$. In particular, participants enjoyed being able to develop code templates in Java and the structural clarity of CrySL. In the post-study interview, all but one participant preferred CogniCrypt$_{GEN}$ over CogniCrypt$_{OLD\text{-}GEN}$. This preference further reflects how significantly more usable CogniCrypt$_{GEN}$ is compared to CogniCrypt$_{OLD\text{-}GEN}$.

Although participants generally enjoyed using CogniCrypt$_{GEN}$, there is still room for improvement—something that is underlined by the post-study interviews. Participants proposed several enhancements to the CogniCrypt$_{GEN}$ API (e.g., abandoning the call-chain design of fluent APIs, shortening the API method names, and providing content assist for class names in the `considerCrySLRule()` call). Three participants also suggested to give the code template a different name from the generated class and to add a comment to `templateUsage()` that indicates it was generated.

5.5 Discussion

Putting everything together, CogniCrypt$_{GEN}$ proves to fulfil the goals we set in the design phase. With CogniCrypt$_{GEN}$, we were capable of implementing eleven common cryptographic use cases, all of which are more compact than with CogniCrypt$_{OLD\text{-}GEN}$. Our experiments have shown that CogniCrypt$_{GEN}$ does not take longer than ten seconds for any of the eleven use cases with negligible memory overhead. Our user-study participants appreciated CogniCrypt$_{GEN}$ significantly more than CogniCrypt$_{OLD\text{-}GEN}$, but requested more proper documentation and made several suggestions to further improve the tool's usability.

5.6 Threats to Validity

Our experiments exhibit several threats to validity. Since we selected graduate students as study participants, the internal validity of the study is threatened as students are not necessarily cryptography experts. Therefore, they may take longer for a given task, because they lack the knowledge for a particular cryptographic API and not as a result of the code generator. We mitigated this threat in two ways. First, the task descriptions included the cryptographic APIs and methods that participants were asked to use. Participants were only asked to implement this solution into the given code generator. Second, we also asked participants to rate their cryptography experience on a 1–10 scale. On average, participants rated themselves at 5.2, with the median self-ascribed experience level of 5. While self-ratings come with their own caveats [1, 16, 25, 31], we did not find statistically significant differences between participants who rated themselves higher than average with those lower than average.

The unequal familiarity of participants with CrySL and XSL threatens the internal validity of the study results as well. When asked to self-rate their experience with each on a 1–10 scale, CrySL scored an average of 5.2, while XSL only reached 1.3. To mitigate this threat, we (1) gave an introduction to both code generators in the beginning and (2) designed the study tasks such that the involvement of CrySL was minimal and modifications to CrySL could be made by anyone who had followed the introduction part of the study. Indeed, we found no significant correlation between the knowledge of CrySL and liking CogniCrypt$_{GEN}$ or being more effective and efficient with it.

In terms of external validity, the relatively low number of participants poses a threat. We addressed this threat by choosing participants from diverse backgrounds regarding experience with Java, Eclipse, and cryptography. We have, in addition, chosen statistical tests appropriate for the number of participants that indeed showed statistical significance.

6 Related Work

Leaving CogniCrypt$_{OLD\text{-}GEN}$ aside, no previous work aims at avoiding cryptographic misuses by generating secure implementations of cryptographic use cases. However, there is indeed work that has attempted to address the widespread misuse through other means. Other work has also presented approaches for generating security code based on specifications. In the following we discuss both kinds of work.

6.1 Detecting and Fixing Cryptographic Misuse through Program Analysis

Previous research on preventing cryptographic misuse has focused on program analysis and repair. Program analysis tools for this purpose [3, 6, 7, 30, 37, 40] are usually equipped with a set of hard-coded rules which they then use to search for rule violations. CryptoLint [7] comes with six rules that address misuses related to encryption. FixDroid [30] and CryptoGuard [37] detect misuses of TLS APIs. Crypto Misuse Analyzer [40] expands on CryptoLint's rule set such that it also includes misuses of hashing, TLS, and key management. CogniCrypt$_{SAST}$ [22] is the firs tool that does support the supply of external specifications through CrySL rules. CogniCrypt's existing rule set [20] covers all cryptographic services of the JCA, ranging from encryption, key management, hashing, mac-ing, and digital signing. Similar tools for other security APIs have been proposed as well [8, 13, 15, 34].

Two tools go beyond detecting misuses and attempt to fix them as well. CDRep [24] applies a by-one-rule-extended version of CryptoLint to a target program. For each of the seven kinds of misuses CryptoLint finds, the authors have devised a fix template. In a second phase, CDRep applies this fix by instrumenting the program's bytecode. In an evaluation on 8,592 Android apps, the tool manages to repair around 95% of the misuses it has detected. FireBugs [41] follows a similar goal. The tool's authors have defined code patterns that contain API misuses. Bootstrapped with these patterns, FireBugs analyzes a target program through program slicing, and repairs it using a series of AST operations. To finally apply a patch, FireBugs employs aspect-oriented programming to weave it into the target program.

In contrast, CogniCrypt$_{GEN}$ does not analyze the target program for misuses. Instead, it uses CrySL rules and code templates to generate secure code into the target program. While its support is limited to the implemented use cases, it prevents misuses from happening in the first place, thus complementing the analysis-based approaches.

6.2 Generating Secure Code

Code generators aiming to produce secure code amount to a huge body of research. There are, for example, code-generation approaches for implementations of cryptographic algorithms [17], security controllers [26], and security protocols [32, 36]. However, only Kane et al. [19] suggest an approach that specifically addresses cryptographic misuses. The authors do not devise and implement a use-case-based code generator like CogniCrypt$_{GEN}$. Instead, they implement several high-level cryptographic protocols like Kerberos or TLS on top of existing low-level cryptographic APIs in Python. Effectively, their protocol implementations are wrapper code similar to what CogniCrypt$_{GEN}$ generates. The two approaches differ in (1) the use cases they support (high-level protocols vs. common cryptographic use cases) and (2) the way use cases may be implemented (hard-coded vs. generated through declarative specifications).

CogniCrypt$_{OLD\text{-}GEN}$ is the closest tool to CogniCrypt$_{GEN}$. CogniCrypt$_{OLD\text{-}GEN}$ combines an algorithm model in the variability-modelling language Clafer [18] with hard-coded XSL templates. Using a constraint solver, CogniCrypt$_{OLD\text{-}GEN}$ fetches secure algorithms from the model. Through a wizard in CogniCrypt, users may configure soluctions for the eight supported cryptographic use cases. CogniCrypt$_{OLD\text{-}GEN}$ stores this user input as well as the selected algorithms in an XML file and uses it to resolve the variability points in the corresponding XSL code template through an XSL transformation. CogniCrypt$_{GEN}$ trumps CogniCrypt$_{OLD\text{-}GEN}$ in terms of usability by facilitating code templates to be in Java. Its code templates are also smaller and, by definition, provably secure with respect to to CrySL specifications – a property hard-coded templates cannot provide. However, our study also revealed a steeper learning curve for CogniCrypt$_{GEN}$.

7 Conclusion

In this work, we presented CogniCrypt$_{GEN}$, a code-generation tool for cryptographic APIs that facilitates the generation of secure and compilable code through code templates and CrySL rules. We have integrated CogniCrypt$_{GEN}$ into the development environment Eclipse to facilitate its use in everyday development. To this end, CogniCrypt$_{GEN}$ covers the most common cryptographic use cases, executes in a few seconds regardless of the use case, and can easily be run on a typical workstation. Our evaluation also revealed low maintenance effort and generally high usability ratings from participants of our user study, especially compared to an XSL-based solution that implements similar use cases.

In future work, we plan to improve the usability of the fluent API. Participants in our user study criticized that class-name parameters are specified as strings instead of, for example, enumerations. They have also suggested to use shorter API-method names and requested more proper documentation. We are grateful for their insights and intend to improve CogniCrypt$_{GEN}$, accordingly. Thanks to CogniCrypt$_{GEN}$, there is now a code-generation engine that allows for implementing more cryptographic use cases. We plan to implement more use cases for other APIs in CogniCrypt$_{GEN}$ and, if necessary, extend its expressiveness.

A Artifact Appendix

A.1 Abstract

In this artefact, we present CogniCrypt$_{GEN}$, a code generation approach that allows for the generation of functionally correct, type-safe, and secure Java code that implements common use cases of cryptographic APIs. To implement a given use case, CogniCrypt$_{GEN}$ requires two artefacts: a) a set of API-usage rules in the specification language CrySL and b) a Java code template specifying which CrySL rules are to be used and how.

The artefact comes with an Eclipse environment, in which CogniCrypt$_{GEN}$ may be executed with all eleven use cases from the original paper. It further contains the artefacts to all use cases to allow for modification and extension. We finally include a tutorial on how CogniCrypt$_{GEN}$ is used.

A.2 Artifact Check-List (Meta-Information)

- **Program: Java bytecode, Java, Java libraries**
- **Compilation: Java 1.8**
- **Data set: CogniCrypt$_{GEN}$ artefacts of eleven common cryptographic use cases**
- **Run-time environment: Eclipse 2019-06 in Java 1.8**
- **Output: Java Code implementing cryptographic use cases**
- **Experiments: RQ1 to RQ3 of CogniCrypt$_{GEN}$ research paper**
- **How much disk space required (approximately)?: 6.4 GB**
- **How much time is needed to prepare workflow (approximately)?: Fifteen minutes**
- **How much time is needed to complete experiments (approximately)?: Ten minutes.**
- **Publicly available?: Yes.**
- **Code licenses (if publicly available)?: EPL 2.0**
- **Data licenses (if publicly available)?: EPL 2.0**

A.3 Description

A.3.1 How Delivered

The artefact is available at https://zenodo.org/record/3597725. We supply the artefact as a VirtualBox image (.ova). CogniCrypt$_{GEN}$ itself may be cloned from Github: https://github.com/eclipse-cognicrypt/CogniCrypt

Both the artefact as well as CogniCrypt$_{GEN}$ are provided under EPL 2.0 license.

A.3.2 Hardware Dependencies

There are no extraordinary hardware dependencies.

A.3.3 Software Dependencies

To run the artefact, the virtualisation software VirtualBox (https://www.virtualbox.org/) is needed. Virtualbox is available for all common Desktop computer operating systems.

A.3.4 Data Sets

Along with the executables for CogniCrypt$_{GEN}$, we provide the artefacts for the eleven use cases that were part of our evaluation (see Table 1 on Page 9 of the original research paper). This, on the one hand, includes the CrySL [22] rule set for the Java Cryptography Architecture(JCA) [20]. By means of the usage specifications in these rules, CogniCrypt$_{GEN}$ determines which methods from a given class are to be called in which order and how instances of different classes may be chained (i.e., they define def-use chains). All rules, we provide with this artefact are also publicly available in their respective latest version at https://github.com/CROSSINGTUD/Crypto-API-Rules/tree/master/JavaCryptographicArchitecture/src.

We also provide the Java code templates that first and foremost specify which classes are required to implement the given use case. As cryptographic use cases may not necessarily solely consist of code relating to cryptography (e.g., file handling in PBE on Files), code templates usually define glue code directly in Java.

In Section A.7, we explain where these artefacts can be found within the artefact and how they may be modified.

A.4 Installation

To run the artefact, first download VirtualBox for your operating system of choice from https://www.virtualbox.org/wiki/Downloads and install it on your machine. Second, you need to download the artefact from https://zenodo.org/record/3597725. Launch VirtualBox and import the artefact through clicking "File" and selecting "Import Applicance". In the file-selection window that pops up, navigate to the location of the artefact and select it.

After VirtualBox has imported the image, launch the VM in VirtualBox. When the VM has booted, log into the user account "cognicrypt" using the password "cognicrypt".

A.5 Experiment Workflow

The artefact facilitates replication of RQ1 to RQ3 of the original evaluation. Please note that we conducted the original evaluation not in a VM, but instead a regular laptop with specs as described in Section 5.2 of the research paper. The results for RQ2 regarding performance and RQ3 regarding memory consumption that the artefact produces may therefore deviate from the original numbers.

In the VM environment, first launch Eclipse by double-clicking the Eclipse icon on the desktop. We provide the implementation of CogniCrypt$_{GEN}$ within this Eclipse's workspace. To then run CogniCrypt$_{GEN}$, launch the run configuration "CogniCrypt". You may do so by selecting the downwards arrow next to the green play button in the in the toolbar and selecting "CogniCrypt" in the menu. This launches

another Eclipse. To distinguish the two Eclipse environemnts, we will hereafter refer to the former as Eclipse and the latter as CogniCrypt.

In CogniCrypt, we provide a test project with a class `Main`, which you may generate code into. To run CogniCrypt$_{GEN}$, click on the CogniCrypt icon, which is located in the tool bar, right under the "Refactor" menu entry. A wizard pops up that allows you to select any of the eleven use cases from Table 1 in RQ1 (i.e. PBE on Files) by clicking on the corresponding button. Click on "Next" and, on the following page, select class *Main*. By clicking the "Generate" button, CogniCrypt$_{GEN}$ is triggered and generates the code for the chosen use case. The generation may take a few seconds. The wizard closes once CogniCrypt$_{GEN}$ has finished.

To run the experiments for RQ2 and RQ3, measure time or memory consumption, respectively. To facilitate time measurements, the wizard contains a checkbox, the enabling of which triggers the same time-measurement mechanism we have used in the original evaluation. The result is shown in both the error log of CogniCrypt *and* the command line of Eclipse.

A.6 Evaluation and Expected Result

After CogniCrypt$_{GEN}$ has completed the code generation, you may find the result in the test project. CogniCrypt$_{GEN}$ has generated two pieces of code. First, in the package `de.crypto.cognicrypt`, it has generated the implementation of the use case. Assuming PBE on Files was chosen, CogniCrypt$_{GEN}$ has generated a class `SecureEncryptor`. This class implements the use case through its three methods. The first one, `getKey()`, generates a key based on the password it receives as an argument. The other two implement en- and decryption of `java.io.File` objects. The second piece of code that CogniCrypt$_{GEN}$ has generated is a single method `templateUsage()` whose purpose is to showcase how to use the implementation, i.e., in the case of PBE on Files, the three aforementioned methods. CogniCrypt$_{GEN}$ generates this method into the class that was selected on the file-selection page of the wizard.

You may also run the generated code by calling the `templateUsage()` method in the `main()` method of the Main class. To simplify running the generated code, we have enriched the `templateUsage()` method with variables that may serve as arguments for the call. We provide further, in comments, how the method should be called, depending on each use case.

A.7 Experiment Customization

You may customize the experiment in three ways. First, the artefact enables you to modify CogniCrypt$_{GEN}$'s implementation in Eclipse any way you like.

Second, you may change the templates for each of the use cases. To do so, navigate to the package `java.de.cognicrypt.codegenerator.crysl.templates` in the plugin **de.cognicrypt.codegenerator** in Eclipse. In this package, you can find the code templates and edit them as regular Java files. For a detailed discussion of the templates and their content, we refer to Section 3.2 in the paper.

Third, you may also modify the CrySL rules, CogniCrypt$_{GEN}$ makes use of. Within the artefact, the he rules are available in *resources/CrySLRules* in the plugin **de.cognicrypt.core**. For a detailed discussion of components of CrySL, we refer to Section 2.2 in the research paper and Krüger et al. [22].

Acknowledgments

This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1119 - 236615297, by the Natural Sciences and Engineering Research Council of Canada, and the Heinz Nixdorf Foundation. We would like to thank Lisa Nguyen Quang Do for her help with the statistics and André Sonntag for his support with the paper-accompanying artefact.

References

[1] Alyce S. Adams, Stephen B. Soumerai, Jonathan Lomas, and Dennis Ross-Degnan. 1999. Evidence of self-report bias in assessing adherence to guidelines. *International Journal for Quality in Health Care* 11, 3 (06 1999), 187–192.

[2] Eric Bodden, Stefan Krueger, Johannes Spaeth, Karim Ali, and Mira Mezini. 2018. CVE-2018-12240. Available from Symantec, CVE-ID CVE-2018-12240.. https://support.symantec.com/us/en/article.SYMSA1460.html

[3] Alexandre Melo Braga, Ricardo Dahab, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. 2017. Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*. 170–181.

[4] John Brooke. 1996. SUS-A quick and dirty usability scale. *Usability Evaluation in Industry* 189, 194 (1996), 4–7.

[5] National Cyber Security Centre. 2018. *Password administration for system owners.* Technical Report. National Cyber Security Centre.

[6] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. 2016. Evaluation of Cryptography Usage in Android Applications. In *International Conference on Bio-inspired Information and Communications Technologies*. 83–90.

[7] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *ACM Conference on Computer and Communications Security*. 73–84.

[8] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. 2012. Why Eve and Mallory love Android: an Analysis of Android SSL (In)security. In *ACM Conference on Computer and Communications Security*. 50–61.

[9] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 121–136.

[10] German Federal Office for Information Security (BSI). 2017. *Cryptographic Mechanisms: Recommendations and Key Lengths.* Technical Report BSI TR-02102-1. BSI.

[11] Martin Fowler. 2005. FluentInterface. https://martinfowler.com/bliki/FluentInterface.html.

[12] Lei Gao. 2005. Latin Squares in Experimental Design. (2005).

[13] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Conference on Computer and Communications Security (CCS)*. 38–49.

[14] Paul Grassi, Michael Garcia, and James Fenton. 2017. *Digital identity guidelines*. Technical Report. National Institute of Standards and Technology.

[15] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, V. N. Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. 2015. Vetting SSL Usage in Applications with SSLINT. In *IEEE Symposium on Security and Privacy*. 519–534.

[16] James R Hebert, Lynn Clemow, Lori Pbert, Ira S Ockene, and Judith K Ockene. 1995. Social desirability bias in dietary self-report may compromise the validity of dietary intake measures. *International journal of epidemiology* 24, 2 (1995), 389–398.

[17] Ekawat Homsirikamol and Kris Gaj. 2014. Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study. In *2014 International Conference on ReConFigurable Computing and FPGAs, ReConFig14, Cancun, Mexico, December 8-10, 2014*. 1–8.

[18] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2019. Clafer: Lightweight Modeling of Structure, Behaviour, and Variability. *Programming Journal* 3, 1 (2019), 2.

[19] Christopher Kane, Bo Lin, Saksham Chand, and Yanhong A. Liu. 2018. High-level Cryptographic Abstractions. *CoRR* abs/1810.09065 (2018).

[20] Stefan Krueger, Johannes Spaeth, Karim Ali, Eric Bodden, and Mira Mezini. 2019. CrySL Rule Set for JCA. https://github.com/CROSSINGTUD/Crypto-API-Rules.

[21] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: Supporting Developers in Using Cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 931–936.

[22] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*.

[23] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why does cryptographic software fail?: a case study and open problems. In *ACM Asia-Pacific Workshop on Systems (APSys)*. 7:1–7:7.

[24] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*. 711–722.

[25] Khalid Mahmood. 2016. Do people overestimate their information literacy skills? A systematic review of empirical evidence on the Dunning-Kruger effect. *Communications in Information Literacy* 10, 2 (2016), 3.

[26] Fabio Martinelli and Ilaria Matteucci. 2012. A framework for automatic generation of security controller. *Softw. Test., Verif. Reliab.* 22, 8 (2012), 563–582.

[27] Kai Mindermann and Stefan Wagner. 2018. Usability and Security Effects of Code Examples on Crypto APIs. In *16th Annual Conference on Privacy, Security and Trust, PST 2018, Belfast, Northern Ireland, Uk, August 28-30, 2018*. 1–2. https://www.cryptoexamples.com/index.html.

[28] Sarah Nadi and Stefan Krüger. 2016. Variability Modeling of Cryptographic Components: Clafer Experience Report. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, Salvador, Brazil, January 27 - 29, 2016*. 105–112.

[29] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: why do Java developers struggle with cryptography APIs?. In *International Conference on Software Engineering (ICSE)*. 935–946.

[30] Duc-Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. 2017. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 1065–1077.

[31] David A Northrup. 1997. *The problem of the self-report in survey research*. Institute for Social Research, York University.

[32] Rick Nunes-Vaz, Steven Lord, and Jolanta Ciuk. 2011. A More Rigorous Framework for Security-in-Depth. *Journal of Applied Security Research* 6, 3 (2011), 372–393.

[33] Philippe Oechslin. 2003. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. 617–630.

[34] Lucky Onwuzurike and Emiliano De Cristofaro. 2015. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. 15:1–15:6.

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

[36] Davide Pozza, Riccardo Sisto, and Luca Durante. 2004. Spi2Java: Automatic Cryptographic Protocol Java Code Generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004), 29-31 March 2004, Fukuoka, Japan*. 400–405.

[37] Sazzadur Rahaman, Ya Xiao, Ke Tian, Fahad Shaon, Murat Kantarcioglu, and Danfeng Yao. 2018. CryptoGuard: Deployment-quality Detection of Java Cryptographic Vulnerabilities. *CoRR* abs/1806.06881 (2018).

[38] Siegfried Rasthofer, Steven Arzt, Robert Hahn, Max Kohlhagen, and Eric Bodden. 2015. (In)Security of Backend-as-a-Service. In *BlackHat Europe 2015*.

[39] Frederick F Reichheld. 2003. The one number you need to grow. *Harvard Business Review* 81, 12 (2003), 46–55.

[40] Shuai Shao, Guowei Dong, Tao Guo, Tianchang Yang, and Chenjie Shi. 2014. Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *nternational Conference on Dependable, Autonomic and Secure Computing*. 75–80.

[41] Larry Singleton, Rui Zhao, Myoungkyu Song, and Harvey P. Siy. 2019. FireBugs: finding and repairing bugs with security patterns. In *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2019, Montreal, QC, Canada, May 25, 2019*. 30–34.