CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs

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

Johannes Späth
Fraunhofer IEM
johannes.spaeth@iem.fraunhofer.de

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

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

Mira Mezini
Technische Universität Darmstadt, Germany
mezini@cs.tu-darmstadt.de

Abstract

Various studies have empirically shown that the majority of Java and Android apps misuse cryptographic libraries, causing devastating breaches of data security. It is crucial to detect such misuses early in the development process. To detect cryptography misuses, one must first define secure uses, a process mastered primarily by cryptography experts, and not by developers.

In this paper, we present CrySL, a definition language for bridging the cognitive gap between cryptography experts and developers. CrySL enables cryptography experts to specify the secure usage of the cryptographic libraries that they provide. We have implemented a compiler that translates such CrySL specification into a context-sensitive and flow-sensitive demand-driven static analysis. The analysis then helps developers by automatically checking a given Java or Android app for compliance with the CrySL-encoded rules.

We have designed an extensive CrySL rule set for the Java Cryptography Architecture (JCA), and empirically evaluated it by analyzing 10,000 current Android apps. Our results show that misuse of cryptographic APIs is still widespread, with 95% of apps containing at least one misuse. Our easily extensible CrySL rule set covers more violations than previous special-purpose tools with hard-coded rules, with our tooling offering a more precise analysis.

2012 ACM Subject Classification Security and privacy → Software and application security, Software and its engineering → Software defect analysis, Software and its engineering → Syntax & Semantics

Keywords and phrases cryptography, domain-specific language, static analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.10

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 10; pp. 10:1–10:28
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Digital devices are increasingly storing sensitive data, which is often protected using cryptography. However, developers must not only use secure cryptographic algorithms, but also *securely* integrate such algorithms into their code. Unfortunately, prior studies suggest that this is rarely the case. Lazar et al. [22] examined 269 published cryptography-related vulnerabilities. They found that 223 are caused by developers misusing a security library while only 46 result from faulty library implementations. Egele et al. [13] statically analyzed 11,748 Android apps using cryptography-related application interfaces (Crypto APIs) and found 88% of them violated at least one basic cryptography rule. Chatzikonstantinou et al. [12] reached a similar conclusion by analyzing apps manually and dynamically. In 2017, VeraCode listed insecure uses of cryptography as the second-most prevalent application-security issue right after information leakage [11]. Such pervasive insecure use of Crypto APIs leads to devastating vulnerabilities such as data breaches in a large number of applications. Rasthofer et al. [31] showed that *virtually all* smartphone apps that rely on cloud services use hard-coded keys. A simple decompilation gives adversaries access to those keys and to all data that these apps store in the cloud.

Nadi et al. [27] were the first to investigate why developers often struggle to use Crypto APIs. The authors conducted four studies, two of which survey Java developers familiar with the Java Crypto APIs. The majority of participants (65%) found their respective Crypto APIs hard to use. When asked why, participants mentioned the API level of abstraction, insufficient documentation without examples, and an API design that makes it difficult to understand how to properly use the API. A potential long-term solution is to redesign the APIs such that they provide an easy-to-use interface for developers that is secure by default. However, it remains crucial to detect and fix the existing insecure API uses. When asked about what would simplify their API usage, participants wished they had tools that help them automatically detect misuses and suggest possible fixes [27]. Unfortunately, approaches based solely on specification inference and anomaly detection [34] are not viable for Crypto APIs, because—as elaborated above—most uses of Crypto APIs are insecure.

Previous work has tried to detect misuses of Crypto APIs through static analysis. While this is a step in the right direction, existing approaches are insufficient for several reasons. First, these approaches implement mostly lightweight *syntactic checks*, which yield fast analysis times at the cost of exposing a high number of false negatives. Therefore, such analyses fail to warn about many insecure (especially non-trivial) uses of cryptography. For instance, applications using password-based encryption commonly do not clear passwords from heap memory and instead rely on garbage collection to free the respective memory space. Moreover, existing tools cannot easily be extended to cover those rules; instead they have cryptography-specific usage rules *hard coded*. The Java Cryptography Architecture (JCA), the primary cryptography API for Java applications [27], offers a plugin design that enables different providers to offer different crypto implementations through the same API, often imposing slightly different usage requirements on their clients. Hard-coded rules can hardly possibly reflect this diversity.

In this paper, we present CrySL, a definition language that enables cryptography experts to specify the secure usage of their Crypto APIs in a lightweight special-purpose syntax. We also present a CrySL compiler that parses and type-checks CrySL rules and translates them into an efficient, yet precise flow-sensitive and context-sensitive static data-flow analysis. The analysis automatically checks a given Java or Android app for compliance with the encoded CrySL rules. CrySL was specifically designed for (and with the help of) cryptography experts. Our approach goes beyond methods that are useful for general validation of API usage (e.g., typestate analysis [3, 7, 28, 8] and data-flow checks [2, 5]) by enabling the expression of domain-specific constraints related to cryptographic algorithms and their parameters.

To evaluate CrySL, we built the most comprehensive rule set available for the JCA classes and interfaces to date, and encoded it in CrySL. We then used the generated static analysis CogniCrypt$_{SAST}$ to scan 10,000 Android apps. We have also modelled the existing hard-coded rules by Egele et al. [13] in CrySL and compared the findings of the generated static analysis (CogniCrypt$_{CL}$) to those of CogniCrypt$_{SAST}$. Our more comprehensive rule set reports 3× more violations, most of which are true warnings. With such comprehensive rules, CogniCrypt$_{SAST}$ finds at least one misuse in 95% of the apps. CogniCrypt$_{SAST}$ is also highly efficient: for more than 75% of the apps, the analysis finishes in under 3 minutes per app, where most of the time is spent in Android-specific call graph construction.

In summary, this paper presents the following contributions:

- We introduce CrySL, a definition language to specify correct usages of Crypto APIs.
- We encode a comprehensive specification of correct usages of the JCA in CrySL.
- We present a CrySL compiler that translates CrySL rules into a static analysis to find violations in a given Java or Android app.
- We empirically evaluate CogniCrypt$_{SAST}$ on 10,000 Android apps.

We have integrated CogniCrypt$_{SAST}$ into crypto assistant CogniCrypt [20] and have open-sourced our implementation and artifacts on GitHub. CogniCrypt$_{SAST}$ is available at `https://github.com/CROSSINGTUD/CryptoAnalysis`. The latest version of the CrySL rules for the JCA can be accessed at `https://github.com/CROSSINGTUD/Crypto-API-Rules`.

2 Related Work

Before we discuss the details of our approach, we contrast it with the following related lines of work: approaches for specifying API (mis)uses, approaches for inferring API specifications, and previous approaches for detecting misuses of security APIs. Our review of these approaches shows that existing specification languages are not optimally suited for defining misuses of Crypto APIs. Additionally, automated inference of correct uses of Crypto APIs is hard to achieve, and existing tools for detecting misuses of Crypto APIs are limited mainly because they have hard-coded rule sets, and support for the most part lightweight syntactic analyses.

2.1 Languages for Specifying and Checking API Properties

There is a significant body of research on textual specification languages that ensure API properties by means of static data-flow analysis. Tracematches [3] were designed to check typestate properties defined by regular expressions over runtime objects. Bodden et al. [8, 10] as well as Naeem and Lhoták [28] present algorithms to (partially) evaluate state matches prior to the program execution, using static analysis.

Martin et al. [24] present Program Query Language (PQL) that enables a developer to specify patterns of event sequences that constitute potentially defective behaviour. A dynamic analysis (i.e., tracematches optimized by a static pre-analysis) matches the patterns against a given program run. A pattern may include a fix that is applied to each match by dynamic instrumentation. PQL has been applied to detecting security-related vulnerabilities such as memory leaks [24], SQL injection and cross-site scripting [23]. Compared to tracematches, PQL captures a greater variety of pattern specifications, at the disadvantage of using only flow-insensitive static optimizations. PQL serves as the main inspiration for the CrySL syntax. Other languages that pursue similar goals include PTQL [16], PDL [26], and TS4J [9].

We investigated tracematches and PQL in detail, yet found them insufficiently equipped for the task at hand. First, both systems follow a black-list approach by defining and finding incorrect program behaviour. We initially followed this approach for crypto-usage mistakes but quickly discovered that it would lead to long, repetitive, and convoluted misuse-definitions. Consequently, CrySL defines desired behaviour, which in the case of Crypto APIs leads to more compact specifications. Second, the above languages are general-purpose languages for bug finding, which causes them to miss features essential to define secure usages of Crypto APIs in particular. The strong focus of CrySL on cryptography allows us to cover a greater portion of cryptography-related problems in CrySL compared to other languages, while at the same time keeping CrySL relatively simple. Third, the CrySL compiler generates state-of-the-art static analyses that were shown to have better performance and precision than other approaches [37], lowering the threat of false warnings.

2.2 Inference/Mining of API-usage specifications

As an alternative to specifying API-usage properties manually, one can attempt to infer them from existing program code. Robillard et al. [33] surveyed over 60 approaches to API property inference. As this survey shows, however, all but two of the surveyed approaches infer patterns from client code (i.e., from applications that use the API in question). When it comes to Crypto APIs, however, past studies have shown that the majority of existing usages of those APIs is, in fact, insecure [13, 12, 35]. Another idea that appears sensible at first sight is to infer correct usage of Crypto APIs from posts on developer portals such as StackOverflow. However, recent studies show that the “solutions” posted there often include insecure code [1].

In result, one can only conclude that automated mining of API-usage specifications is very challenging for Crypto APIs, if it is possible at all. In the future, we plan to investigate a semi-automated approach in which we use automated inference to infer at least partial specifications, but directly in CrySL, that security experts can then further correct and complete by hand.

2.3 Detecting Misuses of Security APIs

Only few previous approaches specifically address the detection of misuses of *security* APIs. CryptoLint [13] performs a lightweight syntactic analysis to detect violations of exactly six hard-coded usage rules for the JCA in Android apps. Those six rules, while important to obey for security, resemble only a tiny fraction of the rule set we provide in this work. It is also hard to specify and validate new rules using CryptoLint, because they would have to be hard-coded. Unlike CryptoLint, CrySL is designed to allow crypto experts to also express comprehensive and complex rules with ease. In Section 8, we extensively compare our tool CogniCrypt$_{\text{SAST}}$ to CryptoLint.

Another tool that finds misuses of Crypto APIs is Crypto Misuse Analyzer (CMA) [35]. Similar to CryptoLint, CMA's rules are hard-coded, and its static analysis is rather basic.

```
1 SecretKeyGenerator kG = KeyGenerator.getInstance("AES");
2 kG.init(128);
3 SecretKey cipherKey = kG.generateKey();
4
5 String plaintextMSG = getMessage();
6 Cipher ciph = Cipher.getInstance("AES/GCM");
7 ciph.init(Cipher.ENCRYPT_MODE, cipherKey);
8 byte[] cipherText = ciph.doFinal(plaintextMSG.getBytes("UTF-8"));
```

Figure 1 An example illustrating the use of `javax.crypto.KeyGenerator` to implement data encryption in Java.

Many of CMA's hard-coded rules are also contained in the CrySL rule set that we provide. Unlike CogniCrypt$_{\text{SAST}}$, CMA has been evaluated on a small dataset of only 45 apps.

Chatzikonstantinou et al. [12] manually identified misuses of Crypto APIs in 49 apps and then verified their findings using a dynamic checker. All three studies concluded that at least 88% of the studied apps misuse at least one Crypto API.

None of the previous approaches facilitates rule creation by means of a higher-level specification language. Instead, the rules are hard-coded into each tool, making it hard for non-experts in static analysis to extend or alter the rule set, and impossible to share rules among tools. Moreover, such hard-coded rules are quite restricted, causing the tools to have a very low recall (i.e., missing many actual API misuses). CrySL, on the other hand, due to its Java-like syntax, enables cryptography experts to easily define new rules. The CrySL compiler then automatically transforms those rules into appropriate, highly-precise static-analysis checks. By defining crypto-usage rules in CrySL instead of hard-coding them, one also makes those rules reusable in different contexts.

3 An Example of a Secure Usage of Crypto APIs

Throughout the paper, we will use the code example in Figure 1 to motivate the language features in CrySL. The code in this figure constitutes an API usage that according to the current state of cryptography research can be considered secure. Lines 1–3 generate a 128-bit secret key to use with the encryption algorithm AES. Lines 5–7 use that key to initialize a Java `Cipher` object that encrypts `plaintextMSG`. Since AES encrypts plaintext block by block, it must be configured to use one of several *modes of operation*. The mode of operation determines how to encrypt a block based on the encryption of the preceding block(s). Line 6 configures `Cipher` to use the Galois/Counter Mode (`GCM`) of operation [25].

Although the code example may look straightforward, a number of subtle alterations to the code would render the encryption non-functional or even insecure. First, both `KeyGenerator` and `Cipher` only support a limited choice of encryption algorithms. If the developer passes an unsupported algorithm to either `getInstance` methods, the respective line will throw a runtime exception. Similarly, the design of the APIs separates the classes for key generation and encryption. Therefore, the developer needs to make sure they pass the same algorithm (here `"AES"`) to the `getInstance` methods of `KeyGenerator` and `Cipher`. If the developer does not configure the algorithms as such, the generated key will not fit the encryption algorithm, and the encryption will fail by throwing a runtime exception. None of the existing tools discussed in Section 2.3 are capable of detecting such functional misuses. Moreover, some supported algorithms are no longer considered secure (e.g., DES or AES/ECB [15]). If the developer selects such an algorithm, the program will still run to completion, but the resulting encryption could easily be broken by attackers. To make

METHOD :=
    methname(PARAMETERS)

PARAMETERS :=
    varname **,** PARAMETERS
    varname

TYPES :=
    QualifiedClassName **,** TYPES
    TYPE

CONSTANTLIST :=
    constant **,** CONSTANTLIST
    constant

AGGREGATE :=
    label | AGGREGATE
    label **;**

EVENT :=
    AGGREGATE
    label **:** METHOD
    label **:** varname **=** METHOD   *A: B = C(D) — a single event with label A consisting of method C, its parameter D, and return object B*

PREDICATE :=
    predname(PARAMETERS)
    predname(PARAMETERS) after EVENT

PREDICATES :=
    PREDICATE **;** PREDICATES

Figure 2 Basic CrySL syntax elements.

things worse, the JCA, the most popular API, offers the insecure ECB mode by default (i.e., when developers request only `"AES"` without specifying a mode of operation explicitly).

To use Crypto APIs properly, developers generally have to take into consideration two dimensions of correctness: (1) the functional correctness that allows the program to run and terminate successfully and (2) the provided security guarantees. Prior empirical studies have shown that developers, for instance by looking for code examples on web portals such as StackOverflow [14], frequently succeed in obtaining functionally correct code. However, they often fail to obtain a secure use of Crypto APIs, primarily because most code examples on those web portals provide "solutions" that themselves are insecure [14].

```
SPEC TYPE;

OBJECTS
    OBJECTS :=
        OBJECT ; OBJECTS                    A ; B — a list of objects A and B
        OBJECT ;                            A — a list of the single object A
    OBJECT :=
        TYPE varname                        A B — object B of Java type A

EVENTS
    EVENTS :=
        EVENT ; EVENTS                      A ; B — a list of events A and B
        EVENT ;                             A — a list of the single event A

FORBIDDEN
    FMETHODS :=
        FMETHOD ; FMETHODS                  A ; B — a list of forbidden A and B
        FMETHOD ;                           A — a list of the single forbidden method A
    FMETHOD :=
        methname(TYPES) => label            A(B) => C — a forbidden method named A
                                            with parameter of Type B and replacement C

ORDER
    USAGEPATTERN :=
        USAGEPATTERN , USAGEPATTERN         A , B — A followed by B
        USAGEPATTERN | USAGEPATTERN         A | B — A or B
        USAGEPATTERN ?                      A? — A is optional
        USAGEPATTERN *                      A* — 0 or more As
        USAGEPATTERN +                      A+ — 1 or more As
        ( USAGEPATTERN )                    (A) — grouping
        AGGREGATE

CONSTRAINTS
    CONSTRAINTS :=
        CONSTRAINT ; CONSTRAINTS
        CONSTRAINT => CONSTRAINT            A => B — A implies B
        CONSTRAINT
    CONSTRAINT :=
        varname in { CONSTANTLIST }         A in {1, 2} — A should be 1 or 2

REQUIRES
    REQ_PREDICATES :=
        PREDICATES

ENSURES
    ENS_PREDICATES :=
        PREDICATES

NEGATES
    NEG_PREDICATES :=
        PREDICATES
```

Figure 3 CrySL rule syntax in Extended Backus-Naur Form (EBNF) [6].

4 CrySL Syntax

As we discuss in Section 2.2, mining API properties for Crypto APIs is extremely challenging, if possible at all, due to the overwhelming number of misuses one finds in actual applications. Hence, instead of relying on the security of existing usages and examples, we here follow an approach in which cryptography experts define correct API usages manually in a special-purpose language, CrySL. In this section, we give an overview of the CrySL syntax elements. A formal treatment of the CrySL semantics is presented in Section 5. Figure 2 presents the basic syntactic elements of CrySL, and Figure 3 presents the full syntax for CrySL rules. Figure 4 shows an abbreviated CrySL rule for `javax.crypto.KeyGenerator`.

4.1 Design Decisions Behind CrySL

We designed CrySL specifically with crypto experts in mind, and in fact with the help of crypto experts. This work was carried out in the context of a large collaborative research center than involves more than a dozen research groups involved in cryptography research. As a result of the domain research conducted within this center, we made the following design decisions when designing CrySL.

White listing. During our domain analysis, we observed that, for the given Crypto APIs, there are many ways they can be misused, but only a few that correspond to correct and secure usages. To obtain concise usage specifications, we decided to design CrySL to use white listing in most places (i.e., defining secure uses explicitly, while implicitly assuming all deviations from this norm to be insecure).

Typestate and data flow. When reviewing potential misuses, we observed that many of them are related to data flows and typestate properties [38]. Such misuses occur because developers call the wrong methods on the API objects at hand, call them in an incorrect order or miss to call the methods entirely. Data-flow properties are important when reasoning about how certain data is being used (e.g., passwords, keys or seed material).

String and integer constraints. In the crypto domain, string and integer parameters are ubiquitously used to select or parametrize specific cryptography algorithms. Strings are widely used, because they are easily recognizable, configurable, and exchangeable. However, specifying an incorrect string parameter may result in the selection of an insecure algorithm or algorithm combination. Many APIs also use strings for user credentials. Those credentials, passwords in particular, should not be hard-coded into the program's bytecode. A precise specification of correct crypto uses must therefore comprise constraints over string and integer parameters.

Tool-independent semantics. We equipped CrySL with a tool-independent semantics (to be presented in Section 5). In the future, those semantics will enable us and others to build other or more effective tools for working with CrySL. For instance, in addition to the static analysis the CrySL compiler derives from the semantics within this paper, we are currently working on a dynamic checker to identify and mitigate CrySL violations at runtime.

Our desire to allow crypto experts to easily express secure crypto uses also precludes us from using existing generic definition languages such as Datalog. Such languages, or minor extensions thereof, might have sufficient expressive power. However, following discussions with crypto developers, we had to acknowledge that they are often unfamiliar with those languages' concepts. CrySL thus deliberately only includes concepts familiar to those developers, hence supporting an easy understanding. We next explain the elements that a typical CrySL rule comprises.

4.2 Mandatory Sections in a CrySL Rule

To provide simple and reusable constructs, a CrySL rule is defined on the level of individual classes. Therefore, the rule starts off by stating the class that it is defined for.

In Figure 4, the **OBJECTS** section defines three objects[1] to be used in later sections of the rule (e.g., the object `algorithm` of type `String`). These objects are typically used as parameters or return values in the **EVENTS** section.

The **EVENTS** section defines all methods that may contribute to the successful use of a `KeyGenerator` object, including two *method event patterns* (Lines 17–18). The first pattern matches calls to `getInstance(String algorithm)`, but the second pattern actually matches calls to two overloaded `getInstance` methods:

- `getInstance(String algorithm, Provider provider)`
- `getInstance(String algorithm, String provider)`

The first parameter of all three methods is a `String` object whose value states the algorithm that the key should be generated for. This parameter is represented by the previously defined `algorithm` object. Two of the `getInstance` methods are overloaded with two parameters. Since we do not need to specify the second parameter in either method, we substitute it with an underscore that serves as a placeholder in one combined pattern definition (Line 18). This concept of method event patterns is similar to pointcuts in aspect-oriented programming languages such as AspectJ [19]. For CrySL, we resort to a more lightweight and restricted syntax as we found full-fledged pointcuts to be unnecessarily complex. Subsequently, the rule defines patterns for the various `init` methods that set the proper parameter values (e.g., keysize) and a `generateKey` method that completes the key generation and returns the generated key.

Line 30 defines a usage pattern for `KeyGenerator` using the keyword **ORDER**. The usage pattern is a regular expression of method event patterns that are defined in **EVENTS**. Although each method pattern defines a label to simplify referencing related events (e.g., `g1`, `i2`, and `GenKey`), it is tedious and error-prone to require listing all those labels again in the **ORDER** section. Therefore, CrySL allows defining *aggregates*. An aggregate represents a disjunction of multiple patterns by means of their labels. Line 19 defines an aggregate `GetInstance` that groups the two `getInstance` patterns. Using aggregates, the usage pattern for `KeyGenerator` reads: there must be exactly one call to one of the `getInstance` methods, optionally followed by a call to one of the `init` methods, and finally a call to `generateKey`.

Following the keyword **CONSTRAINTS**, Lines 33–35 define the constraints for objects defined under **OBJECTS** and used as parameters or return values in the **EVENTS** section. In the abbreviated CrySL rule in Figure 4, the first constraint limits the value of `algorithm` to `"AES"` or `"Blowfish"`. For each algorithm, there is one constraint that restricts the possible values of `keysize`.

The **ENSURES** section is the final mandatory construct in a CrySL rule. It allows CrySL to support rely/guarantee reasoning. The section specifies predicates to govern interactions between different classes. For example, a `Cipher` object uses a key obtained from a

[1] As the example shows, in CrySL, **OBJECTS** also comprise primitive values.

```
SPEC javax.crypto.KeyGenerator

OBJECTS
  java.lang.String algorithm;
  int keySize;
  javax.crypto.SecretKey key;

EVENTS
  g1: getInstance(algorithm);
  g2: getInstance(algorithm, _);
  GetInstance := g1 | g2;

  i1: init(keySize);
  i2: init(keySize, _);
  i3: init(_);
  i4: init(_, _);
  Init := i1 | i2 | i3 | i4;

  GenKey: key = generateKey();

ORDER
  GetInstance, Init?, GenKey

CONSTRAINTS
  algorithm in {"AES", "Blowfish"};
  algorithm in {"AES"} => keySize in {128, 192, 256};
  algorithm in {"Blowfish"} => keySize in {128, 192, 256, 320, 384,
      448};

ENSURES
  generatedKey[key, algorithm];
```

Figure 4 CrySL rule for using `javax.crypto.KeyGenerator`.

```
SPEC javax.crypto.Cipher

OBJECTS
  int encmode;
  java.security.Key key;
  java.lang.String transformation;
  ...

EVENTS
  g1: getInstance(transformation);
  ...
  i1: init(encmode, key);

...

REQUIRES
  generatedKey[key, alg(transformation)];

ENSURES
  encrypted[cipherText, plainText];
```

Figure 5 CrySL rule for using `javax.crypto.Cipher`.

Table 1 Helper Functions in CrySL.

Function	**Purpose**
alg(*transformation*) mode(*transformation*) padding(*transformation*)	Extract algorithm/mode/padding from `transformation` parameter of `Cipher.getInstance` call.
length(*object*)	Retrieve length of *object*
nevertypeof(*object*, *type*)	Forbid *object* to be of *type*
callTo(*method*)	Require call to *method*
noCallTo(*method*)	Forbid call to *method*

`KeyGenerator`. The **`ENSURES`** section specifies what a class guarantees, presuming that the object is used properly. For example, the `KeyGenerator` CrySL rule in Figure 4 ends with the definition of a *predicate* `generatedKey` with the generated key object and its corresponding algorithm as parameters. This predicate may be *required* (i.e., relied on) by the rule for `Cipher` or other classes that make use of such a key through the optional element of the **`REQUIRES`** block as illustrated in Figure 5.

To obtain the required expressiveness, we have further enriched CrySL with some simple built-in auxiliary functions. For example, in Figure 5, the function `alg` extracts the encryption algorithm from `transformation` (Line 55). This function is necessary, because `generatedKey` expects only the encryption algorithm as its second parameter, but `transformation` optionally specifies also the mode of operation and padding scheme (e.g., Line 6 in Figure 1). For instance, `alg` would extract `"AES"` from `"AES/GCM"` or from `"AES/CBC/PKCS5Padding"`. Table Table 1 lists all of these functions. Note the last two functions `callTo` and `noCallTo` may seem redundant to the **`ORDER`** and **`FORBIDDEN`** (see Section 4.3) sections because they appear to fulfil the same purpose of requiring or forbidding certain method calls. However, these two functions go beyond that because they allow for the specification of conditional forbidden and required methods.

4.3 Optional Sections in a CrySL Rule

A CrySL rule may contain optional sections that we showcase through the CrySL rule for `PBEKeySpec`. In Figure 6, the **`FORBIDDEN`** section specifies methods that must *not* be called, because calling them is always insecure. `PBEKeySpec` derives cryptographic keys from a user-given password. For security reasons, it is recommended to use a cryptographic salt for this operation. However, the constructor `PBEKeySpec(char[] password)` does not allow for a salt to be passed, and the implementation in the default provider does not generate one. Therefore, this constructor should not be called, and any call to it should be flagged. Consequently, the CrySL rule for `PBEKeySpec` lists it in the **`FORBIDDEN`** section (Line 72). In the case of `PBEKeySpec`, there is an alternative secure constructor (Line 68). CrySL allows one to specify an alternative method event pattern using the arrow notation shown in Line 72. With **`FORBIDDEN`** events, CrySL's language design deviates a bit from its usual white-listing approach. We made this choice deliberately to keep specifications concise. Without explicit **`FORBIDDEN`** events, one would have to simulate their effect by explicitly listing all events defined on a given type except the one that ought to be forbidden. This would significantly increase the size of CrySL specifications.

In general, predicates are generated for a particular usage whenever it does not use any **`FORBIDDEN`** events, its regular **`EVENTS`** follow the usage pattern defined in the **`ORDER`** section,

```
59 SPEC javax.crypto.spec.PBEKeySpec
60
61 OBJECTS
62   char[] pw;
63   byte[] salt;
64   int it;
65   int keylength;
66
67 EVENTS
68   create: PBEKeySpec(pw, salt, it, keylength);
69   clear: clearPassword();
70
71 FORBIDDEN
72   PBEKeySpec(char[]) => create;
73   PBEKeySpec(char[],byte[],int) => create;
74
75 ORDER
76   create, clear
77 ...
78
79 ENSURES
80   keyspec[this, keylength] after create;
81
82 NEGATES
83   keyspec[this, _];
```

Figure 6 CrySL rule for `javax.crypto.spec.PBEKeySpec`.

and if the usage fulfils all constraints in the **`CONSTRAINTS`** section of its corresponding rule. `PBEKeySpec`, however, deviates from that standard. The class contains a constructor that receives a user-given password, but the method `clearPassword` deletes that password later, making it no longer accessible to other objects that might use the key-spec. Consequently, a `PBEKeySpec` object fulfils its role after calling the constructor but only until `clearPassword` is called.

To model this usage precisely, CrySL allows one to specify a method-event pattern using the keyword **`after`** (Line 80). If the respective method is called, a predicate is generated. Furthermore, CrySL supports invalidating an existing predicate in the **`NEGATES`** section (Line 83). The last call to be made on a `PBEKeySpec` object is the call to `clearPassword` (Line 76). Additionally, the rule lists the predicate `keySpec[this,_]` within the **`NEGATES`** block. Semantically, the negation of the predicates means the following. A final event in the **`ORDER`** pattern, in this case a call to `clearPassword`, invalidates the previously generated `keyspec` predicate(s) for `this`. Section 5.2.2 presents the formal semantics of predicates.

5 CrySL Formal Semantics

5.1 Basic Definitions

A CrySL rule consists of several sections. The **`OBJECTS`** section comprises a set of typed variable declarations $\mathbb{V}$. In the syntax in Figure 3, each declaration $v \in \mathbb{V}$ is represented by the syntax element `TYPE varname`. $\mathbb{M}$ is the set of all resolved method signatures, where each signature includes the method name and argument types. The **`EVENTS`** section contains elements of the form (m, v), where $m \in \mathbb{M}$ and $v \in \mathbb{V}^*$. We denote the set of all methods referenced in **`EVENTS`** by M. The **`FORBIDDEN`** section lists a set of methods from $\mathbb{M}$ denoted by their signatures; forbidden events cannot bind any variables. The **`ORDER`** section specifies the usage pattern in terms of a regular expression of labels or aggregates that are in M, i.e., over the defined **EVENTS**. We express this regular expression formally by the equivalent non-deterministic finite automaton (Q, M, δ, q_0, F) over the alphabet M, where Q is a set of states, q_0 is its initial state, F is the set of accepting states, and $\delta : Q \times M \to \mathcal{P}(Q)$ is the state transition function.

The **CONSTRAINTS** section is a subset of $\mathbb{C} := (\mathbb{V} \to \mathcal{O} \cup \mathcal{V}) \to \mathbb{B}$ (i.e., each constraint is a boolean function), where the argument is itself a function that maps variable names in $\mathbb{V}$ to objects in $\mathcal{O}$ or values with primitive types in $\mathcal{V}$.

A CrySL rule is a tuple $(T, \mathcal{F}, \mathcal{A}, \mathcal{C})$, where T is the reference type specified by the **SPEC** keyword, $\mathcal{F} \subseteq \mathbb{M}$ is the set of forbidden events, $\mathcal{A} = (Q, M, \delta, q_0, F) \in \mathbb{A}$ is the automaton induced by the regular expression of the **ORDER** section, and $\mathcal{C} \subseteq \mathbb{C}$ is the set of **CONSTRAINTS** that the rule lists. We refer to the set of all CrySL rules as $\mathbb{SPEC}$.

Our formal definition of a CrySL rule does not contain the sections **REQUIRES**, **ENSURES**, and **NEGATES**. Those sections reason about the interaction of predicates, whose formal treatment we discuss in Section 5.2.2.

5.2 Runtime Semantics

Each CrySL rule encodes usage constraints to be validated for all runtime objects of the reference type T stated in its **SPEC** section. We define the semantics of a CrySL rule in terms of an evaluation over a runtime program trace that records all relevant runtime objects and values, as well as all events specified within the rule.

▶ Definition 1 (Event). Let $\mathcal{O}$ be the set of all runtime objects and $\mathcal{V}$ the set of all primitive-typed runtime values. An *event* is a tuple $(m, e) \in \mathbb{E}$ of a method signature $m \in \mathbb{M}$ and an *environment* e (i.e., a mapping $\mathbb{V} \to \mathcal{O} \cup \mathcal{V}$ of the parameter variable names to concrete runtime objects and values). If the environment e holds a concrete object for the `this` value, then it is called the event's *base object*.

▶ Definition 2 (Runtime Trace). A *runtime trace* $\tau \in \mathbb{E}^*$ is a finite sequence of events $\tau_0 \ldots \tau_n$.

▶ Definition 3 (Object Trace). For any $\tau \in \mathbb{E}^*$, a subsequence $\tau_{i_1} ... \tau_{i_n}$ is called an *object trace* if $i_1 < ... < i_n$ and all base objects of τ_{i_j} are identical.

Lines 1–2 in Figure 1 result in an object trace that has two events:

$$(m_0, \{algorithm \mapsto \texttt{"AES"}, \texttt{this} \mapsto o_{kg}\})$$
$$(m_1, \{algorithm \mapsto \texttt{"AES"}, keySize \mapsto \texttt{128}, \texttt{this} \mapsto o_{kg}\})$$

where m_0 and m_1 are the signatures of the `getInstance` and `init` methods of the `KeyGenerator` class. For static factory methods such as `getInstance`, we assume that `this` is bound to the returned object. We use o_{kg} to denote that the object `o` is bound to the variable `kG` at runtime.

The decision whether a runtime trace τ satisfies a set of CrySL rules involves two steps. In the first step, individual object traces are evaluated independently of one another. Yet, different runtime objects may still interact with each other. CrySL rules capture this interaction by means of rely/guarantee reasoning, implemented through predicates that a rule ensures on a runtime object. These interactions between different objects are checked against the specification in a second step by considering the predicates they require and ensure. We first discuss individual object traces in more detail.

$$
\begin{aligned}
sat^o &: \mathbb{E}^* \times \mathbb{SPEC} \rightarrow \mathbb{B} \\
[\tau^o, (T^o, \mathcal{F}^o, \mathcal{A}^o, \mathcal{C}^o)] &\rightarrow sat^o_F(\tau^o, \mathcal{F}^o) \ \wedge \\
& \qquad sat^o_{\mathbb{A}}(\tau^o, \mathcal{A}^o) \ \wedge \\
& \qquad sat^o_{\mathbb{C}}(\tau^o, \mathcal{C}^o)
\end{aligned}
$$

Figure 7 The function sat^o verifies an individual object trace for the object o.

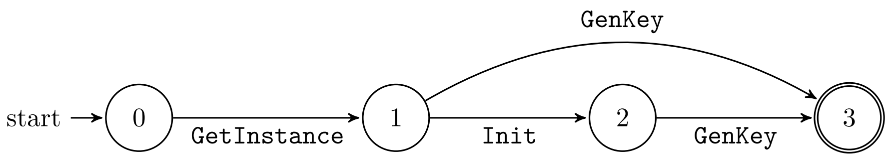

Figure 8 The state machine for the CRYSL rule in Figure 4 (without an implicit error state).

5.2.1 Individual Object Traces

The sections **`FORBIDDEN`**, **`ORDER`** and **`CONSTRAINTS`** are evaluated on individual object traces. Figure 7 defines the function sat^o that is true if and only if a given trace τ^o for a runtime object o satisfies its CRYSL rule. This definition of sat^o ignores interactions with other object traces. We will discuss later how such interactions are resolved. In the following, we assume the trace $\tau^o = \tau^o_0, ..., \tau^o_n$, where $\tau^o_i = (m^o_i, e^o_i)$. To illustrate the computation, we will also refer to our example from Figure 1 and the involved rules of `KeyGenerator` (Figure 4) and `Cipher` (Figure 5). The function sat^o is composed of three sub-functions:

5.2.1.1 Forbidden Events (sat^o_F)

Given a trace τ^o and a set of forbidden events $\mathcal{F}$, sat^o ensures that none of the trace events is forbidden.

$$sat^o_F(\tau^o, \mathcal{F}^o) := \bigwedge_{i=0...n} m^o_i \notin \mathcal{F}^o$$

The CRYSL rule for `KeyGenerator` does not list any forbidden methods. Hence, sat^o trivially evaluates to true for object `kG` in Figure 1.

5.2.1.2 Order Errors ($sat^o_{\mathbb{A}}$)

The second function checks that the trace object is used in compliance with the specified usage pattern (i.e., all methods in the rule are invoked in no other than the specified order). Formally, the sequence of method signatures of the object trace $m^o := m^o_0, \ldots, m^o_n$ (i.e., the projection onto the method signatures) must be an element of the language $\mathcal{L}(\mathcal{A}^o)$ that the automaton $\mathcal{A}^o = (Q, \mathbb{M}, \delta, q_0, F)$ of the **`ORDER`** section induces. By definition of language containment, after the last observed signature of the trace m^o_n, the corresponding state of the automaton must be an accepting state $s \in F$. This definition ignores any variable bindings. They are evaluated in the second step.

$$sat^o_{\mathbb{A}}(\tau^o, \mathcal{A}^o) := m^o \in \mathcal{L}(\mathcal{A}^o)$$

Figure 8 displays the automaton created for `KeyGenerator` using the aggregate names as labels. State *0* is the initial state, and state *3* is the only accepting state. Following the code in Figure 1 for the object `kG` of type `KeyGenerator`, the automaton transitions from state *0* to *1* at the call to `getInstance` (Line 1). With the calls to `init` (Line 2) and `generateKey` (Line 3), the automaton first moves to state *2* and finally to state *3*. Therefore, function $sat^o_{\mathbb{A}}$ evaluates to true for this example.

5.2.1.3 Constraints ($sat^o_{\mathbb{C}}$)

The validity check of the constraints ensures that all constraints of $\mathcal{C}$ are satisfied. This check requires the sequence of environments $(e^o_0, ..., e^o_n)$ of the trace τ^o. All objects that are bound to the variables along the trace must satisfy the constraints of the rule.

$$sat^o_{\mathbb{C}}(\tau^o, \mathcal{C}^o) := \bigwedge_{c \in \mathcal{C}^o, i=0 \ldots n} c(e^o_i)$$

To compute $sat^o_{\mathbb{C}}$ for the `KeyGenerator` object `kG` at the call to `getInstance` in Line 1, only the first constraint has to be checked. This is because the corresponding environment e^o_1 holds a value only for `algorithm`, and the other two constraints reference other variable names. The evaluation function c returns true if `algorithm` assumes either `"AES"` or `"Blowfish"` as its value, which is the case in Figure 1. The computation of $sat^o_{\mathbb{C}}$ for Lines 2–3 works similarly.

5.2.2 Interaction of Object Traces

To define interactions between individual object traces, the **`REQUIRES`**, **`ENSURES`**, and **`NEGATES`** sections allow individual CrySL rules to reference one another. For a rule for one object to hold at any given point in an execution trace, all predicates that its **`REQUIRES`** section lists must have been both previously *ensured* (by other specifications) and not *negated*. Predicates are *ensured* (i.e., generated) and *negated* (i.e., killed) by certain events. Formally, a predicate is an element of $\mathbb{P} := \{(\mathit{name}, \mathit{args}) \mid \mathit{args} \in \mathbb{V}^*\}$ (i.e., a pair of a predicate name and a sequence of variable names). Predicates are generated in specific states. Each CrySL rule induces a function $\mathcal{G} \colon S \to \mathcal{P}(\mathbb{P})$ that maps each state of its automaton to the predicate(s) that the state generates.

The predicates listed in the **`ENSURES`** and **`NEGATES`** sections may be followed by the term **`after`** n, where n is a method event pattern label or aggregate. The states that follow the event or aggregate n in the automaton generate the respective predicate. If the term **`after`** is not used for a predicate, the final states of the automaton generate (or negate) that predicate (i.e., we interpret it as **`after`** n, where n is an event that leads to a final state).

In addition to states selected as predicate-generating, the predicate is also ensured if the object resides in any state that transitively follows the selected state, unless the states are explicitly (de-)selected for the same predicate within the **`NEGATES`** section. At any state that generates a predicate, the event driving the automaton into this state binds the variable names to the values that the specification previously collected along its object trace.

Formally, an event $n^o = (m^o, e^o) \in \mathbb{E}$ of a rule r and for an object o ensures a predicate $p = (\mathit{predName}, \mathit{args}) \in \mathbb{P}$ on the objects $e^o \in \mathcal{O}$ if:

1. The method m^o of the event leads to a state s of the automaton that generates the predicate p (i.e., $p \in \mathcal{G}(s)$).
2. The runtime trace of the event's base object o satisfies the function sat^o.

```
84 boolean option1 = isPrime(66); //some non-trivial predicate returning
       false
85 byte[] input = "Message".getBytes("UTF-8");
86
87 String alg = "SHA-256";
88 if (option1) alg = "MD5";
89 MessageDigest md = MessageDigest.getInstance(alg);
90
91 if (input.size() > 0) md.update(input);
92 byte[] digest = md.digest();
```

Figure 9 An example illustrating the usage of `java.security.MessageDigest` in Java.

3. All relevant **REQUIRES** predicates of the rule are satisfied at execution of event n^o.

For the `KeyGeneraor` object `kG` in Figure 1, a predicate is generated at Line 7 because (1) its automaton transitions to its only predicate-generating state (state *3* of the automaton in Figure 8), (2) sat^o evaluates to true as previously shown for each subfunction and (3) the corresponding CrySL rule does not require any predicates.

6 Detecting Misuses of Crypto APIs

To detect all possible rule violations, our tool CogniCrypt$_{\text{SAST}}$ approximates the evaluation function sat^o using a static data-flow analysis. In a security context, it is a requirement to detect as many misuses as possible. One drawback is the potential for false warnings that originate from over-approximations any static analysis requires. In the following, we use the example in Figure 9 to illustrate why and where approximations are required. We will show later in our evaluation that, in practice, our analysis is highly precise and that the chosen approximations rarely actually lead to false warnings.

The code example in Figure 9 implements a hashing operation. By default, the code uses `SHA-256`. However, if the condition `option1` evaluates to true, `MD5` is chosen instead (Line 88). The CrySL rule for `MessageDigest`, displayed in Figure 10, does not allow the usage of `MD5` though, because it is no longer secure [15].

The `update` operation is performed only on non-empty input (Line 91). Otherwise, the call to `update` is skipped and only the call to `digest` is executed, without any input. Although not strictly insecure, this usage does not comply with the CrySL rule for `MessageDigest`, because it leads to no content being hashed.

To approximate sat^o_F, the analysis must search for possible forbidden events by first constructing a call graph for the whole program under analysis. It then iterates through the graph to find calls to forbidden methods. Depending on the precision of the call graph, the analysis may find calls to forbidden methods that cannot be reached at runtime.

The analysis represents each runtime object o by its allocation site. In our example, allocation sites are `new` expressions and calls to `getInstance` that return an object of a type for which a CrySL rule exists. For each such allocation site, the analysis approximates $sat^o_{\mathbb{A}}$ by first creating a finite-state machine. CogniCrypt$_{\text{SAST}}$ then evaluates the state machine using a typestate analysis that abstracts runtime traces by program paths. The typestate analysis is path-insensitive, thus, at branch points, it assumes that both sides of the branch may execute. In our contrived example, this feature leads to a false positive: although the condition in Line 91 always evaluates to true, and the call to `update` is never actually skipped, the analysis considers that this may happen, and thus reports a rule violation.

```
SPEC java.security.MessageDigest

OBJECTS
  java.lang.String algorithm;
  byte[] input;
  int offset;
  int length;
  byte[] hash;
  ...

EVENTS
  g1: getInstance(algorithm);
  g2: getInstance(algorithm, _);
  Gets := g1 | g2;
  ...
  Updates := ...;

  d1: output = digest();
  d2: output = digest(input);
  d3: digest(hash, offset, length);
  Digests := d1 | d2 | d3;

  r: reset();

ORDER
  Gets, (d2 | (Updates+, Digests)), (r, (d2 | (Updates+, Digests)))*

CONSTRAINTS
  algorithm in {"SHA-256", "SHA-384", "SHA-512"};

ENSURES
  digested[hash, ...];
  digested[hash, input];
```

Figure 10 CrySL rule for `java.security.MessageDigest`.

To approximate $sat^o_{\mathbb{C}}$, we have extended the typestate analysis to also collect potential runtime values of variables along all program paths where an allocated object is used. The constraint solver first filters out all *irrelevant* constraints. A constraint is irrelevant if it refers to one or more variables that the typestate analysis has not encountered. In Figure 10, the rule only includes one internal constraint—on variable `algorithm`. If we add a new internal constraint to the rule about the variable `offset`, the constraint solver will filter it out as irrelevant when analyzing the code in Figure 9 because the only method this variable is associated with (`digest` labelled `d3`) is never called. The analysis distinguishes between never encountering a variable in the source code and not being able to extract the values of a variable. With the same rule and code snippet, if the analysis fails to extract the value for `algorithm`, the constraint evaluates to false. Collecting potential values of a variable over all possible program paths of an allocation site may lead to further imprecision. In our example, the analysis cannot statically rule out that `algorithm` may be `MD5`. The rule forbids the usage of `MD5`. Therefore, the analysis reports a misuse.

Handling predicates in our analysis follows the formal description very closely. If sat^o evaluates to true for a given allocation site, the analysis checks whether all required predicates for the allocation site have been ensured earlier in the program. In the trivial case, when no predicate is required, the analysis immediately ensures the predicate defined in the **`ENSURES`** section. The analysis constantly maintains a list of all ensured predicates, including the statements in the program that a given predicate can be ensured for. If the allocation site under analysis requires predicates from other allocation sites, the analysis consults the list of ensured predicates and checks whether the required predicate, with matching names and arguments, exists at the given statement. If the analysis finds all required predicates, it ensures the predicate(s) specified in the **`ENSURES`** section of the rule.

7 Implementation

We have implemented the CrySL compiler using Xtext [17], an open-source framework for developing domain-specific languages as well as the CrySL- parameterizable static analysis CogniCrypt$_{\text{SAST}}$. We have further integrated CogniCrypt$_{\text{SAST}}$ with CogniCrypt [20], in which it replaces the original code-analysis component.

7.1 CrySL

Given the CrySL grammar, Xtext provides a parser, type checker, and syntax highlighter for the language. When supplied with a type-safe CrySL rule, Xtext outputs the corresponding AST, which is then used to generate the required static analysis.

We developed CrySL rules for all relevant JCA classes in an iterative process. That is, we first worked through the JCA documentation to produce a set of rules and then refined these rules through selective discussions with cryptographers and searching security blogs and forums. In total, we have devised 23 rules covering classes ranging from key handling to digital signing. All rules define a usage pattern. Some classes (e.g. `IvParameterSpec`) contain one call to a constructor only, while others (e.g. `Cipher`) involve almost ten elements with several layers of nesting. Fifteen rules come with parameter constraints, eight of which contain limitations on cryptographic algorithms. The eight rules without parameter constraints are mostly related to classes whose purpose is to set up parameters for specific encryptions (e.g. `GCMParameterSpec`). All rules define at least one **`ENSURES`** predicate, while only eleven require predicates from other rules. Across all rules, we have only declared two methods forbidden. We do not find this low number surprising as such methods are always insecure and should not at all be part of a security API. If at all, two forbidden methods is too high a number. All rules are available at `https://github.com/CROSSINGTUD/Crypto-API-Rules`.

7.2 CogniCrypt$_{\text{sast}}$

CogniCrypt$_{\text{SAST}}$ consists of several extensions to the program analysis framework Soot [39, 21]. Soot transforms a given Java program into an intermediate representation that facilitates executing intra- and inter-procedural static analyses. The framework provides standard static analyses such as call-graph construction. Additionally, Soot can analyze a given Android app intra-procedurally. Further extensions by FlowDroid [5] enable the construction of Android-specific call graphs that are necessary to perform inter-procedural analysis.

Validating the **ORDER** section in a CrySL rule requires solving the typestate check $sat^{o}_{\mathbb{A}}$. To this end, we use IDE^{al}, a framework for efficient inter-procedural data-flow analysis [37], to instantiate a typestate analysis. The analysis defines the finite-state machine $\mathcal{A}^{o}$ to check against and the allocation sites to start the analysis from. From those allocation sites, IDE^{al} performs a flow-, field-, and context-sensitive typestate analysis.

The constraints and the predicates require knowledge about objects and values associated with rule variables at given execution points in the program. The typestate analysis in CogniCrypt$_{\text{SAST}}$ extracts the primitive values and objects on-the-fly, where the latter are abstracted by allocation sites. When the typestate analysis encounters a call site that is referred to in an event definition, and the respective rule requires the object or value of an argument to the call, CogniCrypt$_{\text{SAST}}$ triggers an on-the-fly backward analysis to extract the objects or values that may participate in the call. This on-the-fly analysis yields comparatively high performance and scalability, because many of the arguments of interest are values of type `String` and `Integer`. Thus, using an on-demand computation avoids constant propagation of *all* strings and integers through the program. For the on-the-fly backward analysis, we extended the on-demand pointer analysis Boomerang [36] to propagate both allocation sites and primitive values. Once the typestate analysis is completed, and all required queries to Boomerang are computed, CogniCrypt$_{\text{SAST}}$ solves the internal constraints and predicates using our own custom-made solvers.

CogniCrypt$_{\text{SAST}}$ may be operated as a standalone command line tool. This way, it takes a program as input and produces an error report detailing misuses and their locations. However, we have further integrated CogniCrypt$_{\text{SAST}}$ into CogniCrypt [20]. CogniCrypt is a Eclipse plugin, which supports developers in using Crypto APIs by means of scenario-based code generation as well code analysis for Crypto APIs. In this context, CogniCrypt translates misuses found by CogniCrypt$_{\text{SAST}}$ into standard Eclipse error markers.

8 Evaluation

We evaluate our implementation CogniCrypt$_{\text{SAST}}$ using the following research questions:

RQ1: What are the precision and recall of CogniCrypt$_{\text{SAST}}$?
RQ2: What types of misuses does CogniCrypt$_{\text{SAST}}$ find?
RQ3: How fast does CogniCrypt$_{\text{SAST}}$ run?
RQ4: How does CogniCrypt$_{\text{SAST}}$ compare to the state of the art?

To answer these questions, we applied the generated static analysis CogniCrypt$_{\text{SAST}}$ to 10,000 Android apps from the AndroZoo dataset [4] using our full CrySL rule set for the JCA. We ran our experiments on a Debian virtual machine with sixteen cores and 64 GB RAM. We chose apps that are available in the official Google Play Store and received an update in 2017. This ensures that we report on the most up-to-date usages of Crypto APIs. We make available all artefacts at this Github repository: `https://github.com/CROSSINGTUD/paper-crysl-reproduciblity-artefacts`.

8.1 Precision and Recall (RQ1)

Setup

To compute precision and recall, the first two authors manually checked 50 randomly selected apps from our dataset for typestate errors and violations of internal constraints. To collect this random sample, we implemented a Java program that generates random numbers using `SecureRandom` and retrieved the apps from the corresponding lines in the spreadsheet containing the results of analysing the 10,000 apps. We did not check for unsatisfied predicates or forbidden events, because these are hard to detect manually—while it may seem simple to check for calls to forbidden events, it is non-trivial to determine whether or not such calls reside in dead code. We compare the results of our manual analysis to those reported by CogniCrypt$_{SAST}$. The goal of this evaluation is to compute precision and recall of the analysis implementation in CogniCrypt$_{SAST}$, not the quality of our CrySL rules. We discuss the latter in Section 8.4. Consequently, we define a false positive to be a warning that should not be reported according to the specified rule, irrespective of that rule's semantic correctness. Similarly, a false negative would arise if CogniCrypt$_{SAST}$ missed to report a misuse that, according to the CrySL rule, does exist in the analyzed program.

Results

In the 50 apps we inspected, CogniCrypt$_{SAST}$ detects 228 usages of JCA classes. Table 2 lists the misuses that CogniCrypt$_{SAST}$ finds (156 misuses in total). In particular, CogniCrypt$_{SAST}$ issues 27 typestate-related warnings, with only 2 false positives. Both arise because the analysis is path-insensitive (Section 6). We further found 4 false negatives that are caused by initializing a `MessageDigest` or a `MAC` object without completing the operation. CogniCrypt$_{SAST}$ fails to find these typestate errors because the supporting off-the-shelf alias analysis Boomerang times out, causing CogniCrypt$_{SAST}$ to abort the typestate analysis without reporting a warning for the object at hand. A larger timeout or future improvements to the alias analysis Boomerang would avoid this problem.

The automated analysis finds 129 constraint violations. We were able to confirm 110 of them. In the other 19 cases, highly obfuscated code causes the analysis to fail to extract possible runtime values statically. For such values, the constraint solver reports the corresponding constraint as violated. A better handling of such highly obfuscated code can be enabled by techniques complementary to ours. For instance, one could augment CogniCrypt$_{SAST}$ with the hybrid static/dynamic analysis Harvester [32]. We have also checked the apps for missed constraint violations (false negatives), but were unable to find any.

> **RQ1**: In our manual assessment, the typestate analysis achieves high precision (92.6%) and recall (86.2%). The constraint resolution has a precision of 85.3% and a recall of 100%.

Table 2 Correctness of CogniCrypt$_{SAST}$ warnings.

	Total Warnings	False Positives	False Negatives
Typestate	27	2	4
Constraints	129	19	0
Total	156	21	4

8.2 Types of Misuses (RQ2)

Setup

We report findings obtained by analyzing all our 10,000 Android apps from AndroZoo [4]. We then use the results of our manual analysis (Section 8.1) as a baseline to evaluate our findings on a large scale.

CogniCrypt$_{SAST}$ detects the usage of at least one JCA class in 8,422 apps. Further investigation unveiled that many of these usages originate from the same common libraries included in the applications. To avoid counting the same crypto usages twice, and to prevent over-counting, we exclude usages within packages `com.android`, `com.facebook.ads`, `com.google` or `com.unity3d` from the analysis.

Results

Excluding the findings in common libraries, CogniCrypt$_{SAST}$ detects the usage of at least one JCA class in 4,349 apps (43% of the analyzed apps). Most of these apps (95%) contain at least one misuse. Across all apps, CogniCrypt$_{SAST}$ started its analysis for a total of 40,295 allocation sites (i.e., abstract objects). Of these, a total of 20,426 individual object traces violate at least one part of the specified rule patterns. CogniCrypt$_{SAST}$ reports typestate errors (**`ORDER`** section in the rule) for 4,708 objects, and reports a total of 4,443 objects to have unsatisfied predicates (i.e., the object expected a predicate from another object as listed in the **`REQUIRES`** block of a rule). The analysis also discovered 97 reachable call sites that call forbidden events. The majority of object traces that violate at least one part of a CrySL rule (54.7%) contradict a constraint listed in the **`CONSTRAINTS`** section of a rule.

Approximately 86% of these constraint-violations are related to `MessageDigest`. In our manual analysis (see RQ1), 89 of the 110 found constraint violations originated from usages of `MD5` and `SHA-1`. We expect a similar fraction to also hold for the 11,178 constraint contradictions reported over all 10,000 apps. Many developers still use `MD5` and `SHA-1`, although both are no longer recommended by security experts [15]. CogniCrypt$_{SAST}$ identifies 1,228 (10.9%) constraint violations related to `Cipher` usages. In our manual analysis, all misuses of the `Cipher` class are due to using the insecure algorithm `DES` or the `ECB` mode of operation. This result is in line with the findings of prior studies [13, 35, 12].

More than 75% of the typestate errors that CogniCrypt$_{SAST}$ issues are caused by misuses of `MessageDigest`. Our manual analysis attributes this high number to incorrect usages of the method `reset()`. In addition to misusing `MessageDigest`, misuses of `Cipher` contribute 766 typestate errors. Finally, CogniCrypt$_{SAST}$ detects 157 typestate errors related to `PBEKeySpec`. The **`ORDER`** section of the CrySL rule for `PBEKeySpec` requires calling `clearPassword()` at the end of the lifetime of a `PBEKeySpec` object. We manually inspected 3 of the misuses and observed that the call to `clearPassword()` is missing in all of them.

Predicates are unsatisfied when CogniCrypt$_{SAST}$ expects the interaction of multiple object traces but is not able to prove their correct interaction. With 4,443 unsatisfied predicates reported, the number may seem relatively large, yet one must keep in mind that unsatisfied predicates accumulate transitively. For example, if CogniCrypt$_{\text{SAST}}$ cannot ensure a predicate for a usage of `IVParameterSpec`, it will not generate a predicate for the key object that `KeyGenerator` generates using the `IVParameterSpec` object. Transitively, CogniCrypt$_{\text{SAST}}$ reports an unsatisfied predicate also for any `Cipher` object that relies on the generated key object.

CogniCrypt$_{\text{SAST}}$ also found 97 calls to forbidden methods. Since only two JCA classes require the definition of forbidden methods in our CrySL rule set (`PBEKeySpec` and `Cipher`), we do not find this low number surprising. A manual analysis of a handful of reports suggests that most of the reported forbidden methods originate from calling the insecure `PBEKeySpec` constructors, as we explained in Section 4.

From the 4,349 apps that use at least one JCA Crypto API, 2,896 apps (66.6%) contain at least one typestate error, 1,367 apps (31.4%) lack required predicates, 62 apps (1.4%) call at least one forbidden method, and 3,955 apps (90.9%) violate at least one internal constraint. Ignoring the class `MessageDigest`, and hereby excluding `MD5` and `SHA-1` constraints, 874 apps still violate at least one constraint in other classes.

RQ2: Approximately 95% of apps misuse at least one Crypto API. Violating the constraints of `MessageDigest` is the most common type of misuse.

8.3 Performance (RQ3)

Setup

CogniCrypt$_{\text{SAST}}$ comprises four main phases. It constructs (1) a *call graph* using FlowDroid [5] and then runs the actual analysis (Section 6), which (2) calls the *typestate analysis* and (3) *constraint analysis* as required, attempting to (4) *resolve all declared predicates*. During the analysis of our dataset, we measured the execution time that CogniCrypt$_{\text{SAST}}$ spent in each phase. We ran CogniCrypt$_{\text{SAST}}$ once per application and capped the time of each run to 30 minutes.

In Section 8.2, we report that CogniCrypt$_{\text{SAST}}$ found usages of the JCA in 4,349 of all 10,000 apps in our dataset. If we include in the reporting those usages that arise from misuses within the common libraries previously excluded (see Section 8.2), this number rises to 8,422. We include the analysis of the libraries in this part of the evaluation because it helps evaluate the general performance of the analysis in the worst case when whole applications are analyzed.

Results

Figure 11 summarizes the distribution of analysis times for the four phases and the total analysis time across these 8,422 apps. For each phase, the box plot highlights the median, the 25% and 75% quartiles, and the minimal and maximal values of the distribution.

Across the apps in our dataset, there is a large variation in the reported execution time (10 seconds to 28.6 minutes). We attribute this variation to the following reasons. The analyzed apps have varying sizes—the number of reachable methods in the call graph varies between 116 and 16,219 (median: 3,125 methods). The majority of the total analysis time (83%) is spent on call-graph construction. For the remaining three phases of the analysis, the distribution is as follows. Across all apps, the resolution of all declared predicates takes

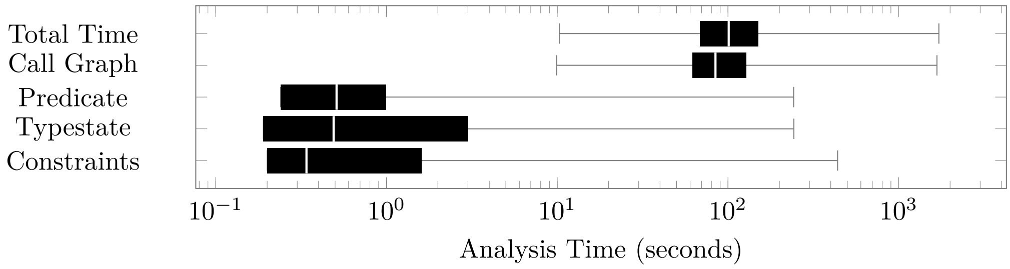

Figure 11 Analysis time (in log scale) of the individual phases of CogniCrypt$_{SAST}$ when running on the apps that use the JCA.

approximately a median of 50 milliseconds, and the typestate analysis phase takes a median of 500 milliseconds. The median for the constraint phase is 350 milliseconds. Therefore, the major bottleneck for the analysis is call-graph construction, a problem orthogonal to the one we address in this work. Our analysis itself is efficient and the overall analysis time is clearly dominated by the runtime of the call-graph construction.

RQ3: On average, CogniCrypt$_{SAST}$ analyzes an app in 101 seconds, with call-graph construction taking most of the time (83%).

8.4 Comparison to Existing Tools (RQ4)

Setup

We compare CogniCrypt$_{SAST}$ to CryptoLint [13], as we explained in Section 2.3 the most closely related tool. Unfortunately, despite contacting the authors we were unable to obtain access to CryptoLint's implementation. We thus resorted to reimplementing the original rules that are hard-coded in CryptoLint as CrySL rules. The fact that all CryptoLint rules can be modelled in CrySL shows its superior expressiveness.

In this section, RuleSet$_{FULL}$ denotes CogniCrypt's comprehensive CrySL rules that we have created for all the JCA classes, while RuleSet$_{CL}$ denotes the set of CrySL rules that we developed to model the original CryptoLint rules. Additionally, CogniCrypt$_{SAST}$ denotes our analysis when it runs using RuleSet$_{FULL}$, and CogniCrypt$_{CL}$ denotes the analysis when it runs using RuleSet$_{CL}$.

RuleSet$_{FULL}$ consists of 23 rules, one for each class of the JCA. RuleSet$_{CL}$ comprises only six individual rules, and they only use the sections **`ENSURES`**, **`REQUIRES`** and **`CONSTRAINTS`**. In other words, the original hard-coded CryptoLint rules do not comprise typestate properties nor forbidden methods. For three out of six rules, we managed to exactly capture the semantics of the hard-coded CryptoLint rule in a respective CrySL rule. The remaining three rules (3, 4, and 6 of the original CryptoLint rules) cannot be perfectly expressed as a CrySL rule, and our CrySL-based rules over-approximate them instead.

CryptoLint rule 4, for instance, requires salts in `PBEKeySpec` to be non-constant. In CrySL, such a relationship is expressed through predicates. Predicates in CrySL, however, follow a white-listing approach and therefore only model correct behaviour. Therefore, in CrySL we model the CryptoLint rule for `PBEKeySpec` in a stricter manner, requiring the salt to be not just non-constant but truly random, i.e., returned from a proper random generator. We followed a similar approach with the other two CryptoLint rules that we modelled in CrySL. In result, $\text{Ruleset}_{\text{CL}}$ is stricter than the original implementation of CryptoLint. In the comparison of $\text{CogniCrypt}_{\text{SAST}}$ and $\text{CogniCrypt}_{\text{CL}}$ in terms of their findings, the stricter rules produce more warnings than the original implementation of CryptoLint. In our comparison against $\text{CogniCrypt}_{\text{SAST}}$, this setup favours CryptoLint because we assume that these additional findings to be true positives. Both rule sets are available at `https://github.com/CROSSINGTUD/Crypto-API-Rules`.

Results

$\text{CogniCrypt}_{\text{CL}}$ detects usages of JCA classes in 1,866 Android apps. For these apps, $\text{CogniCrypt}_{\text{CL}}$ reports 5,507 misuses, only a third of the 20,426 misuses that $\text{CogniCrypt}_{\text{SAST}}$ identifies using $\text{Ruleset}_{\text{FULL}}$, our more comprehensive rule set.

Using $\text{CogniCrypt}_{\text{CL}}$, all reported warnings are related to 6 classes, compared to 23 classes that are specified in $\text{Ruleset}_{\text{FULL}}$. As we have pointed out, CryptoLint does not specify any typestate properties or forbidden methods. Hence, $\text{CogniCrypt}_{\text{CL}}$ does not find the 4,805 warnings that $\text{CogniCrypt}_{\text{SAST}}$ identifies in these categories using $\text{Ruleset}_{\text{FULL}}$. Furthermore, while $\text{CogniCrypt}_{\text{SAST}}$ reports 11,178 constraint violations with the standard rules, $\text{CogniCrypt}_{\text{CL}}$ reports only 1,177 constraint violations. Of the 11,178 constraint violations, 9,958 are due to the rule specification for the class `MessageDigest`. CryptoLint does not model this class. If we remove these violations, 1,609 violations are still reported by $\text{CogniCrypt}_{\text{SAST}}$, a total of 432 more than by $\text{CogniCrypt}_{\text{CL}}$.

We compare our findings to the study by Egele et al. [13] that identifies the use of `ECB` mode as a common misuse of cryptography. In that study, 7,656 apps use `ECB` (65.2% of apps that use Crypto APIs). On the other hand, in our study, $\text{CogniCrypt}_{\text{CL}}$ identified 663 uses of ECB mode in 35.5% of apps that use Crypto APIs. Although a high number of apps still exhibit this basic misuse, there is a considerable decrease (from 65.2% to 35.5%) compared to the previous study by Egele et al. [13]. Given that all apps in our study must have received an update in 2017, we believe that the decrease of misuses reflects taking software security more seriously in today's app development.

Based on the high precision (92.6%) and recall (96.2%) values discussed in **RQ1**, we argue that $\text{CogniCrypt}_{\text{SAST}}$ provides an analysis with a much higher recall than CryptoLint. Although the larger and more comprehensive rule set, $\text{Ruleset}_{\text{FULL}}$, detects more complex misuses, the precise analysis keeps the false-positive rate at a low percentage.

RQ4: The more comprehensive $\text{Ruleset}_{\text{FULL}}$ detects 3× as many misuses as CryptoLint in almost 4× more JCA classes.

8.5 Threats to Validity

Our ruleset $\text{Ruleset}_{\text{FULL}}$ is mainly based on the documentation of the JCA [18]. Although we have significant domain expertise, our CrySL-rule specifications for the JCA are only as correct as the JCA documentation. Our static-analysis toolchain depends on multiple external components and despite an extensive set of test cases, of course, we cannot fully rule out bugs in the implementation.

Java allows a developer to programmatically select a non-default cryptographic service provider. $\text{CogniCrypt}_{\text{SAST}}$ currently does not detect such customizations but instead assumes that the default provider is used. This behaviour may lead to imprecise results because our rules forbid certain default values that are insecure for the default provider, but may be secure if a different one is chosen.

9 Conclusion

In this paper, we present CrySL, a description language for correct usages of cryptographic APIs. Each CrySL rule is specific to one class, and it may include usage pattern definitions and constraints on parameters. Predicates model the interactions between classes. For example, a rule may generate a predicate on an object if it is used successfully, and another rule may require that predicate from an object it uses. We also present a compiler for CrySL that transforms a provided ruleset into an efficient and precise data-flow analysis $\text{CogniCrypt}_{\text{SAST}}$ checking for compliance according to the rules. For ease of use, we have integrated $\text{CogniCrypt}_{\text{SAST}}$ and with Eclipse crypto assistant CogniCrypt. Applying $\text{CogniCrypt}_{\text{SAST}}$, the analysis for our extensive ruleset $\text{Ruleset}_{\text{FULL}}$, to 10,000 Android apps, we found 20,426 misuses spread over 95% of the 4,349 apps using the JCA. $\text{CogniCrypt}_{\text{SAST}}$ is also highly efficient: for more than 75% of the apps the analysis finishes in under 3 minutes, where most of the time is spent in Android-specific call graph construction.

In future work, we plan to address the following challenges. We have developed all the rules used in $\text{CogniCrypt}_{\text{SAST}}$ ourselves. While we have acquired some deeper familiarity with cryptographic concepts in general and the JCA in particular, we are not cryptographers. Therefore, we are open to and want cryptography experts to correct potential mistakes in our existing rules. We would further encourage domain experts to model their own cryptographic libraries in CrySL to improve the support in $\text{CogniCrypt}_{\text{SAST}}$ and, by extension, CogniCrypt. CrySL currently only supports a binary understanding of security—a usage is either secure or not. We would like to enhance CrySL to have a more fine-grained notion of security to allow for more nuanced warnings in $\text{CogniCrypt}_{\text{SAST}}$. This is challenging because the CrySL language still ought to be concise. Additionally, CrySL currently requires one rule per class per JCA provider, because there is no way to express the commonality and variability between different providers implementing the same algorithms, leading to specification overhead. To address this issue, we plan to modularize the language using import and override mechanisms. Moreover, we plan to extend CrySL to support more complex properties such as using the same cryptographic key for multiple purposes. We will also perform consistency checks for the CrySL rules. For now, only Xtext-based type checks are performed.

Lastly, we also intend on applying CrySL in other contexts. One of the authors of this paper has already started to have students implement a dynamic checker to identify and mitigate violations at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle [29] are being used as well and we will to extend $\text{CogniCrypt}_{\text{SAST}}$ to support them. Additionally, we will investigate to which extent CrySL is applicable to Crypto APIs in other programming languages. At the time of writing, we are exploring CrySL's compatibility with OpenSSL [30]. We finally aim to examine whether CrySL is expressive enough to meaningfully specify usage constraints for non-crypto APIs.

Acknowledgments

This work was supported by the DFG through its Collaborative Research Center CROSSING, the project RUNSECURE, by the Natural Sciences and Engineering Research Council of Canada, by the Heinz Nixdorf Foundation, a Fraunhofer ATTRACT grant, and by an Oracle Collaborative Research Award. We would also like to thank the maintainers of AndroZoo for allowing us to use their data set in our evaluation.

References

1 Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl. Developers need support, too: A survey of security advice for software developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26, Sept 2017. 10.1109/SecDev.2017.17.

2 Dima Alhadidi, Amine Boukhtouta, Nadia Belblidia, Mourad Debbabi, and Prabir Bhattacharya. The dataflow pointcut: a formal and practical framework. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009*, pages 15–26, 2009.

3 Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 345–364, 2005. 10.1145/1094811.1094839.

4 Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 468–471, 2016.

5 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269, 2014.

6 John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, Michael Woodger, and Peter Naur. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

7 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 301–320, 2007. 10.1145/1297027.1297050.

8 Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE '10: International Conference on Software Engineering*, pages 5–14, New York, NY, USA, May 2010. ACM. ISBN 978-1-60558-719-6.

9 Eric Bodden. TS4J: a fluent interface for defining and computing typestate analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis, SOAP 2014, Edinburgh, UK, Co-located with PLDI 2014, June 12, 2014*, pages 1:1–1:6, 2014.

10 Eric Bodden, Patrick Lam, and Laurie Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2):7:1–7:52, June 2012.

11 VeraCode (CA). State of software security 2017. `https://info.veracode.com/report-state-of-software-security.html`, 2017.

12 Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. Evaluation of cryptography usage in android applications. In *International Conference on Bio-inspired Information and Communications Technologies*, pages 83–90, 2016.

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

14 Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 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*, pages 121–136, 2017.

15 German Federal Office for Information Security (BSI). Cryptographic mechanisms: Recommendations and key lengths. Technical Report BSI TR-02102-1, BSI, January 2017.

16 Simon Goldsmith, Robert O'Callahan, and Alexander Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 385–402, 2005.

17 Xtext home page. `http://www.eclipse.org/Xtext/`, 2017.

18 Oracle Inc. Java Cryptography Architecture (JCA) Reference Guide. `https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html`, 2017.

19 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. *ECOOP 2001—Object-Oriented Programming*, pages 327–354, 2001.

20 Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 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*, pages 931–936, 2017.

21 Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, October 2011.

22 David Lazar, Haogang Chen, Xi Wang, and Nickolai 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.

23 V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, 2005.

24 Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 365–383, 2005.

25 David A. McGrew and John Viega. The security and performance of the galois/counter mode (GCM) of operation. In *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, pages 343–355, 2004.

26 Clint Morgan, Kris De Volder, and Eric Wohlstadter. A static aspect language for checking design rules. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007*, pages 63–72, 2007.

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

28 Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 347–366, 2008.

29 Legion of the Bouncy Castle Inc. BouncyCastle, 2018. `https://www.bouncycastle.org/java.html`.

30 OpenSSL. OpenSSL - Cryptography and SSL/TLS Toolkit, 2018. `https://www.openssl.org/`.

31 Siegfried Rasthofer, Steven Arzt, Robert Hahn, Max Kohlhagen, and Eric Bodden. (in)security of backend-as-a-service. In *BlackHat Europe 2015*, November 2015.

32 Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Network and Distributed System Security Symposium (NDSS)*, February 2016.

33 Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering (TSE)*, 39:613–637, 2013.

34 Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE TOSEM*, 39(5):613–637, May 2013. ISSN 0098-5589. 10.1109/TSE.2012.63.

35 Shuai Shao, Guowei Dong, Tao Guo, Tianchang Yang, and Chenjie Shi. Modelling analysis and auto-detection of cryptographic misuse in Android applications. In *nternational Conference on Dependable, Autonomic and Secure Computing*, pages 75–80, 2014.

36 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 22:1–22:26, 2016.

37 Johannes Späth, Karim Ali, and Eric Bodden. Ide^{al}: Efficient and precise alias-aware dataflow analysis. In *2017 International Conference on Object-Oriented Programming, Languages and Applications (OOPSLA/SPLASH)*. ACM Press, October 2017. To appear.

38 Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. 10.1109/TSE.1986.6312929. URL `https://doi.org/10.1109/TSE.1986.6312929`.

39 Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction*, pages 18–34, 2000.