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

Stefan Krüger*, Johannes Späth†, Karim Ali‡, Eric Bodden*† and Mira Mezini§

*Paderborn University, Germany, firstname.lastname@upb.de
†Fraunhofer IEM, firstname.lastname@iem.fraunhofer.de
‡University of Alberta, Canada, karim.ali@ualberta.ca
§Technische Universität Darmstadt, Germany, mezini@cs.tu-darmstadt.de

Abstract—Various studies have empirically shown that the majority of Java and Android applications 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 define secure uses first, a process mastered primarily by cryptography experts but not by developers. In this paper, we present CRYSL, a specification language for bridging the cognitive gap between cryptography experts and developers. CRYSL enables cryptography experts to specify the secure usage of the cryptographic libraries 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 and all 204,788 current Java software artefacts on Maven Central. Our results show that misuse of cryptographic APIs is still widespread, with 95% of apps and 63% of Maven artefacts containing at least one misuse. Our easily extensible CRYSL rule set covers more violations than previous special-purpose tools that contain hard-coded rules, while still offering a more precise analysis.

Index Terms—cryptography, domain-specific language, static analysis.

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. [30] 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. [18] statically analyzed 11,748 Android apps using cryptography-related application programming interfaces (Crypto APIs) and found 88% of them violated at least one basic cryptography rule. Chatzikonstantinou et al. [16] 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 [15]. Such pervasive insecure use of Crypto APIs leads to devastating vulnerabilities such as data breaches in a large number of applications. Rasthofer et al. [42] 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. [35] 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 [35]. Unfortunately, approaches based solely on specification inference and anomaly detection [47] are not viable for Crypto APIs, because—as elaborated above—most uses of Crypto APIs are insecure [41].

Previous work has tried to detect misuses of Crypto APIs through static analysis. While this step is 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 missing 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 more complex scenarios; instead they have hard-coded cryptography-specific usage rules. The Java Cryptography Architecture (JCA), the primary cryptography API for Java applications [35], 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 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. CrySL is meant to serve as a building block for different kinds of tool support, including documentation, patch, or use-case-based code generation as well as program analysis. In this work, we further present one such tool, namely COGNICRYPT\textsubscript{SAST}, 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 \cite{2,10,11,35} and data-flow checks \cite{2}) 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\textsubscript{SAST} to conduct two studies. First, we scan 10,000 Android apps. We have also modelled the existing hard-coded rules by Egele et al. \cite{18} in CrySL and compared the findings of the generated static analysis to those of COGNICRYPT\textsubscript{SAST} for the 10,000 Android apps. Our more comprehensive rule set reports 3× more violations, most of which are true warnings. With such comprehensive rules, COGNICRYPT\textsubscript{SAST} finds at least one misuse in 95% of the apps. COGNICRYPT\textsubscript{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 the second study, we apply COGNICRYPT\textsubscript{SAST} to all 204,788 software artefacts on Maven Central, the world’s largest Java code repository, and present the first comprehensive study of misuses of crypto APIs in Java. This study facilitates an investigation into whether there is a difference between average developers for Java and Android in terms of how securely they use cryptographic APIs. We find this matter worthy of investigation as we would assume regular Java code to contain significantly fewer misuses due to the relative maturity of Java as a language and breadth of application fields. Across all analyzed artefacts, COGNICRYPT\textsubscript{SAST} finds 24,349 cryptography misuses in 5,712 Java artefacts. More than 63% of all artefacts that use the JCA contain at least one misuse. We, therefore, conclude that Java code is indeed less insecure, but overall still not secure.

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\textsubscript{SAST} on 10,000 Android apps and all Maven Central software artefacts and, based on the results, draw conclusions on the state of cryptographic application security in Android and Java.

We have integrated COGNICRYPT\textsubscript{SAST} into the Eclipse-based crypto-API assistant COGNICRYPT\textsubscript{SAST} that, among other things, continuously checks JCA-related code for misuses through static analyses. We replaced COGNICRYPT\textsubscript{SAST}’s former static-analysis component with COGNICRYPT\textsubscript{SAST}. To facilitate external contributions, we have also open-sourced our implementation and artefacts on GitHub. COGNICRYPT\textsubscript{SAST} is available at \url{https://github.com/CROSSINGTUD/CryptoAnalysis}. The latest version of the CrySL rules for the JCA can be accessed at \url{https://github.com/CROSSINGTUD/Crypto-API-Rules}. This paper is based on a conference paper \cite{28} published at the European Conference on Object-Oriented Programming 2018.

2 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.

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

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

- 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\textsubscript{SAST} on 10,000 Android apps and all Maven Central software artefacts and, based on the results, draw conclusions on the state of cryptographic application security in Android and Java.

We have integrated COGNICRYPT\textsubscript{SAST} into the Eclipse-based crypto-API assistant COGNICRYPT\textsubscript{SAST} that, among other things, continuously checks JCA-related code for misuses through static analyses. We replaced COGNICRYPT\textsubscript{SAST}’s former static-analysis component with COGNICRYPT\textsubscript{SAST}. To facilitate external contributions, we have also open-sourced our implementation and artefacts on GitHub. COGNICRYPT\textsubscript{SAST} is available at \url{https://github.com/CROSSINGTUD/CryptoAnalysis}. The latest version of the CrySL rules for the JCA can be accessed at \url{https://github.com/CROSSINGTUD/Crypto-API-Rules}. This paper is based on a conference paper \cite{28} published at the European Conference on Object-Oriented Programming 2018.

2 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\textsuperscript{3} generate a 128-bit secret key to use with the encryption algorithm AES. Lines 5\textsuperscript{2} 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 7\textsuperscript{6} configures Cipher to use the Galois/Counter Mode (GCM) of operation \cite{33}.

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() method, 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 9.3 are
capable of detecting such functional misuses. Moreover,
some supported algorithms are no longer considered secure
(e.g., DES or AES/ECB [21]). 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 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 explicit).

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 secu-
rity guarantees. Prior empirical studies have shown that
developers, for instance by looking for code examples on
web portals such as StackOverflow [20], 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 “solu-
tions” that themselves are insecure [20].

3 CRYSL SYNTAX

As we discuss in Section 9.2, 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 4.

3.1 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 that 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 mis-
uses, we observed that many of them are related to data
flows and typestate properties [53]. 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 rec-
ognizable, 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 cre-
dentials. 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 Sec-
9.4). 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. This tool will help us overcome
challenges posed by static analyses, as described in
Section 5.

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 de-
velopers, 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.

The resulting language is not, per se, limited to express-
ing usage constraints on cryptographic APIs. While there are
certain elements in CRYSL, such as the integer and String
constraints, that are more essential to cryptographic than
to other APIs, we do assume the language to be capable
of covering those other APIs as well. We nonetheless view
CRYSL (and COGNICRYPTO) as domain-specific because
we tailored them to the domain of cryptography through
an extensive domain analysis, which resulted in, among
other things, the aforementioned language elements. We
have, however, not conducted an in-depth investigation into
CRYSL’s applicability to other APIs of other domains and
leave this to future work.

Rules in CRYSL are split into multiple sections as a
means to follow the separation-of-concerns paradigm. This
way, required method calls are defined independently of
forbidden ones, constraints on an object may be specified
separately from assigning this object a role as method ar-
gument or return object of a method, and the correct order
of method calls is defined without interference from object
definitions or declarations of forbidden method calls. These
separations improve readability and, as described further
below, facilitate reuse of elements within a single rule. In
early discussions of CRYSL with domain experts, this design
was received positively. We next explain the individual
elements that a typical CRYSL rule comprises by means of
Figure 2, which shows an abbreviated CRYSL rule for
javax.crypto.KeyGenerator.
3.2 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 2, the OBJECTS section defines three objects [3] 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 [26]. 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 [20] 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 listed under OBJECTS and used as parameters or return values in the EVENTS section. In the abbreviated CrySL rule in Figure 2, 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.

1. As the example shows, in CrySL, OBJECTS also comprise primitive values.
TABLE 1  
Helper Functions in CrySL.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>alg(transformation)</td>
<td>Extract algorithm/mode/padding</td>
</tr>
<tr>
<td>mode(transformation)</td>
<td>from transformation parameter</td>
</tr>
<tr>
<td>padding(transformation)</td>
<td>of Cipher.getInstance() call.</td>
</tr>
<tr>
<td>length(object)</td>
<td>Retrieve length of object.</td>
</tr>
<tr>
<td>neverTypeOf(object, type)</td>
<td>Forbid object to be of type.</td>
</tr>
<tr>
<td>callTo(method)</td>
<td>Require call to method.</td>
</tr>
<tr>
<td>noCallTo(method)</td>
<td>Forbid call to method.</td>
</tr>
</tbody>
</table>

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

Fig. 4. CrySL rule for javax.crypto.spec.PBEKeySpec.

the optional element of the REQUIRES block as illustrated in Figure 3.

To obtain the required expressiveness, we have further enriched CrySL with some simple built-in auxiliary functions. For example, in Figure 3 the function alg extracts the encryption algorithm from transformation (Line 59). 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 1 lists all of these functions. Note the last two helper functions callTo and noCallTo may seem redundant to the ORDER and FORBIDDEN (see Section 3.3) sections because they appear to fulfill 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.

3.3 Optional Sections in a CrySL Rule

A CrySL rule may contain optional sections that we showcase through the CrySL rule for PBEKeySpec. In Figure 4, 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. Depending on the tool support, these alternatives may either be used for constructive error messages and documentation, or automated fix generation. 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, and if the usage fulfills 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 fulfills 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). Usually, a predicate is supposed to be generated, when an object of the given type has successfully and fully followed the call pattern given in its ORDER section. However, with the after keyword, a predicate is generated right after the respective method is called. Furthermore, CrySL supports invalidating an existing predicate in the NEGATES section (Line 85). The last call to be made on a PBEKeySpec object is the call to clearPassword() (Line 72). 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 4.2.2 presents the formal semantics of predicates.

For reference, we provide the basic syntactic elements of CrySL and the full syntax in Figures 5 and 6 respectively.

4 CrySL Formal Semantics

CrySL may serve as a basis for multiple kinds of tool support. In this section, we, therefore, present a formal semantics of the language that is tool-independent. For a discussion of our CrySL-based static analysis COGNICRYPTSAST, we refer the reader to Section 5.
4.1 Basic Definitions

A CrySL rule consists of several sections. The **OBJECTS** section comprises a set of typed variable declarations $V$. In the syntax in Figure 5, each declaration $v \in V$ is represented by the syntax element `TYPE varname`. $M$ 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 $m \in M$ and $v \in V^*$. We denote the set of all methods referenced in **EVENTS** by $M$. The **FORBIDDEN** section lists a set of methods from $M$ 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 $(Q, M, \delta, q_0, F)$ 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 $\delta : Q \times M \rightarrow \mathcal{P}(Q)$ is the state transition function.

The **CONSTRAINTS** section is a subset of $\mathcal{C} := (V \rightarrow O \cup V) \rightarrow \mathbb{B}$ (i.e., each constraint is a boolean function), where the argument is itself a function that maps variable names in $V$ to objects in $O$ or values with primitive types in $V$. A CrySL rule is a tuple $(T, F, A, C)$, where $T$ is the reference type specified by the **SPEC** keyword, $F \subseteq M$ is the set of forbidden events, $A = (Q, M, \delta, q_0, F) \in \mathcal{A}$ is the automaton induced by the regular expression of the **ORDER** section, and $C \subseteq \mathcal{C}$ is the set of **CONSTRAINTS** that the rule lists. We refer to the set of all CrySL rules as **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 4.2.2.

4.2 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 $O$ be the set of all runtime objects and $V$ the set of all primitive-typed runtime values. An event is a tuple $(m, e) \in \mathcal{E}$ of a method signature $m \in M$ and an environment $e$ (i.e., a mapping $V \rightarrow O \cup V$ 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 $\tau \in \mathcal{E}^*$ is a finite sequence of events $\tau_0 \ldots \tau_n$.

**Definition 3** (Object Trace). For any $\tau \in \mathcal{E}^*$, a subsequence $\tau_{i_1} \ldots \tau_{i_n}$ is called an object trace if $i_1 < \ldots < i_n$ and all base objects of $\tau_{i_j}$ are identical.
\[ \text{sat}^o : \mathbb{E}^* \times \text{SPEC} \to \mathbb{B} \]
\[ [\tau^o, (T^o, F^o, A^o, C^o)] \to \text{sat}_F^o(\tau^o, F^o) \land \text{sat}_R^o(\tau^o, A^o) \land \text{sat}_C^o(\tau^o, C^o) \]

Fig. 7. The function \text{sat}^o verifies an individual object trace for the object \( o \).

Lines 1,2 in Figure 1 result in an object trace that has two events:
\[
\begin{align*}
(m_0, \{\text{algorithm} \Rightarrow "AES", \text{this} \mapsto o_{kg}\}) \\
(m_1, \{\text{algorithm} \Rightarrow "AES", \text{keySize} \mapsto 128, \text{this} \mapsto o_{kg}\})
\end{align*}
\]

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 \( \tau \) 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.

4.2.1 Individual Object Traces

The sections FORBIDDEN, ORDER and CONSTRAINTS are evaluated on individual object traces. Figure 2 defines the function \text{sat}^o that is true if and only if a given trace \( \tau^o \) for a runtime object \( o \) satisfies its CrySL rule. This definition of \text{sat}^o ignores interactions with other object traces. We will discuss later how such interactions are resolved. In the following, we assume the trace \( \tau^o = \tau_0^o, ..., \tau_n^o \), where \( \tau_i^o = (m_i^o, e_i^o) \). To illustrate the computation, we will also refer to our example from Figure 1 and the involved rules of KeyGenerator (Figure 2) and Cipher (Figure 3). The function \text{sat}^o is composed of three sub-functions:

4.2.1.1 Forbidden Events (\( \text{sat}_F^o \)): Given a trace \( \tau^o \) and a set of forbidden events \( F \), \text{sat}^o ensures that none of the trace events is forbidden.

\[ \text{sat}_F^o(\tau^o, F^o) := \bigwedge_{i=0...n} m_i^o \notin F^o \]

The CrySL rule for KeyGenerator does not list any forbidden methods. Hence, \text{sat}^o trivially evaluates to true for object \( kG \) in Figure 1.

4.2.1.2 Order Errors (\( \text{sat}_R^o \)): 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 \( m^o := m_0^o, ..., m_n^o \) (i.e., the projection onto the method signatures) must be an element of the language \( \mathcal{L}(A^o) \) that the automaton \( A^o = (Q, M, \delta, q_0, F) \) of the ORDER section induces. Therefore, it is

\[ \text{sat}_R^o(\tau^o, A^o) := m^o \in \mathcal{L}(A^o) \]

By definition of language containment, after the last observed signature of the trace \( m_n^o \), 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.

Figure 3 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 \text{sat}^o evaluates to true for this example.

4.2.1.3 Constraints (\( \text{sat}_C^o \)): The validity check of the constraints ensures that all constraints of \( C \) are satisfied. This check requires the sequence of environments \( (e_0^o, ..., e_n^o) \) of the trace \( \tau^o \). All objects that are bound to the variables along the trace must satisfy the constraints of the rule.

\[ \text{sat}_C^o(\tau^o, C^o) := \bigwedge_{c \in C^o, i=0...n} c(e_i^o) \]

To compute \text{sat}^o 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 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 \text{sat}^o for Lines 2,3 works similarly.

4.2.2 Interaction of Object Traces

To define interactions between individual object traces, the \text{REQUIRES}, \text{ENSURES}, and \text{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 \text{REQUIRES} section lists must have been both previously \emph{ensured} (by other specifications) and not \emph{negated}. Predicates are ensured (i.e., generated) and negated (i.e., killed) by certain events. Formally, a predicate is an element of \( \mathbb{P} := \{(\text{name}, \text{args}) \mid \text{args} \in V^*\} \) (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 \( G : S \to \mathbb{P} \) that maps each state of its automaton to the predicate(s) that the state generates.

The predicates listed in the \text{ENSURES} and \text{NEGATES} sections may be followed by the term \text{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 $n^o = (m^o, e^o) \in E$ of a rule $r$ and for an object $o$ ensures a predicate $p = (\text{predName}, \text{args}) \in P$ 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 $sat^o$.
3. All relevant `REQUIRES` predicates of the rule are satisfied at execution of event $n^o$.

For the `KeyGenerator` 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 3), (2) $sat^o$ evaluates to true as previously shown for each subfunction and (3) the corresponding CRYSL rule does not require any predicates.

## 5 Detecting Misuses of Crypto APIs

To detect all possible rule violations, our tool CogniCRYSTAST 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 83). The CRYSL rule for `MessageDigest`, displayed in Figure 10, does not allow the usage of `MD5` though, because it is no longer secure [21].

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. A hash function used without any input does not comply with the CRYSL rule for `MessageDigest`; it is most likely a programming error as no content is being hashed.

To approximate $sat^o_p$, 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.

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

**Fig. 9.** An example illustrating the usage of java.security.MessageDigest in Java.

**Fig. 10.** CRYSL rule for java.security.MessageDigest.

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 $sat^o_p$ by first creating a finite-state machine. CogniCRYSTAST 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.

To approximate $sat^o_p$, 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 \textit{algorithm}. If we add a new internal constraint to the rule about the variable \textit{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 (\textit{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 \textit{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 \textit{algorithm} may be \texttt{MD5}. The rule forbids the usage of \texttt{MD5}. Therefore, the analysis reports a misuse.

Handling predicates in our analysis follows the formal description very closely. If \texttt{sat} 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 \texttt{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 \texttt{ENSURES} section of the rule.

6 IMPLEMENTATION

We have implemented the CRYSL compiler using Xtext [24], an open-source framework for developing domain-specific languages as well as the CRYSL-parameterizable static analysis COGNI\texttt{CRYPT\_AST}. We have further integrated COGNI\texttt{CRYPT\_AST} with COGNI\texttt{CRYPT} [27], in which it replaces the original code-analysis component.

6.1 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. \texttt{IvParameterSpec}) contain one call to a constructor only, while others (e.g. \texttt{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. \texttt{GCMParameterSpec}). All rules define at least one \texttt{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.

6.1.1 Rule Set for the JCA

Apart from the rules we have discussed for \texttt{KeyGenerator} and \texttt{Cipher}, the full rule set of \texttt{COGNICRYPT\_AST}, encompasses a total of 23 CRYSL specifications that specify correct uses of all JCA classes, which offer various cryptographic services. In the following, we describe these services with their respective classes and briefly summarize important usage constraints. All mentioned classes are defined in the packages \texttt{javax.crypto} and \texttt{java.security} of the JCA.

Asymmetric Key Generation: Asymmetric and symmetric cryptography requires different key formats. Asymmetric cryptography uses pairs of public and private keys. While one of the keys encrypts plaintexts to ciphertexts, the second key decrypts the ciphertext. The JCA models a key pair as class \texttt{KeyPair} and are generated by \texttt{KeyPairGenerator}.

Symmetric Key Generation: Symmetric cryptography uses the same key for encryption and decryption. The JCA models symmetric keys as type \texttt{SecretKey}, generated by a \texttt{SecretKeyFactory} or \texttt{KeyGenerator}. The \texttt{SecretKeyFactory} also enables password-based cryptography using \texttt{PBEParameterSpec} or \texttt{PBESecureKey}.

Signing and Verification of Data: The class \texttt{Signature} of the JCA allows one to digitally sign data and verify a signature based on a private/public key pair. A \texttt{Signature} requires the key pair to be correctly generated, hence the rule for \texttt{Signature} requires a predicate from the asymmetric-key generation task.

Generation of Initialization Vectors: Initialization vectors (IVs) are used to add entropy to ciphertexts of encryptions. An IV must have enough randomness and must be properly generated. The JCA class \texttt{IvParameterSpec} wraps a byte array as an IV and it is required for the array to be randomized by \texttt{SecureRandom}. The CRYSL rule for \texttt{IvParameterSpec} requires a predicate randomized.

Encryption and Decryption: The key component of the JCA is represented by the class \texttt{Cipher}, which implements functionality to encrypt or decrypt data. Depending on the used algorithms, modes and paddings must be selected and keys and initialization vectors must be properly generated. Hence, the complete CRYSL rule for \texttt{Cipher} requires many other cryptographic services to be executed securely earlier and list them in its respective \texttt{REQUIRES} clause.
Hashing & MACs: There are two forms of cryptographic hash functions. A MAC is an authenticated hash that requires a symmetric key, but there are also keyless hash functions such as MD5 or SHA-256. The JCA's class Mac implements functionality for mac-ing, while keyless hashes are computed by MessageDigest.

Persisting Keys: Securely storing key material is an important cryptographic task for confidentiality and integrity of the encrypted data. The JCA class KeyStore supports developers in this task and stores the key material. Cryptographically Secure Random-Number Generation: Randomness is vital in all aspects of cryptography. Java offers cryptographically secure pseudo-random number generators through SecureRandom. As discussed for PBESecret, SecureRandom often acts as a helper and therefore many rules list the randomized predicate in their own REQUIRES section.

Combination of Different Cryptographic Services: In practice, cryptographic services are often combined to achieve more security goals than one primitive could offer on its own. One often-used example is so-called authenticated encryption that achieves not only confidentiality, but also authenticity and integrity on the original plaintext. To this end, MACs and encryption are combined. While there are multiple ways to combine the two, only first encrypting the plaintext and then computing the MAC on the ciphertext is recommended [21]. As such combinations of different cryptographic services are implemented through source code as well, we have explicitly encoded secure combinations in the rules of participating classes through predicates.

6.2 CogniCrypt_{SAST}

CogniCrypt_{SAST} consists of several extensions to the program analysis framework Soot [29 55]. 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 [6] 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 \( sat_t^u \). To this end, we use IDE\(^d\), a framework for efficient inter-procedural data-flow analysis [53], to instantiate a typestate analysis. The analysis defines the finite-state machine \( A^e \) to check against and the allocation sites to start the analysis from. From these allocation sites, IDE\(^d\) 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_{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_{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 [51] to propagate both allocation sites and primitive values. Once the typestate analysis is completed, and all required queries to Boomerang are computed, CogniCrypt_{SAST} solves the internal constraints and predicates using our own custom-made solvers.

CogniCrypt_{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. On top of that, we have further integrated CogniCrypt_{SAST} into CogniCrypt [27]. CogniCrypt is an Eclipse plug-in, which supports developers in using Crypto APIs by means of scenario-based code generation as well code analysis for Crypto APIs to find misuses of them. The code generation provides implementations for common cryptographic coding tasks (e.g. file encryption, or establishing secure connections). For misuse detection, we have replaced CogniCrypt’s underlying static-analysis tool TSJ [12] with CogniCrypt_{SAST}. In this context, CogniCrypt translates misuses found by CogniCrypt_{SAST} into standard Eclipse error markers.

7 Crypto-API Misuse in Android Apps

We first evaluate CogniCrypt_{SAST} by addressing the following research questions:

RQ1: What are the precision and recall of CogniCrypt_{SAST}?

RQ2: What types of misuses does CogniCrypt_{SAST} find in Android apps?

RQ3: How fast does CogniCrypt_{SAST} run?

RQ4: How does CogniCrypt_{SAST} compare to the state of the art?

To answer these questions, we applied the generated static analysis CogniCrypt_{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 restriction 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-reproducibility-artefacts

7.1 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
TABLE 2
Correctness of COGNICRYPT SAST warnings.

<table>
<thead>
<tr>
<th>Typestate</th>
<th>False Positives</th>
<th>False Negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>156</td>
<td>21</td>
</tr>
</tbody>
</table>

TABLE 3
Types of API Misuses reported by COGNICRYPT SAST for Android apps that use the JCA.

<table>
<thead>
<tr>
<th>API Misuse Type</th>
<th># Warnings</th>
<th># Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect calling sequences</td>
<td>4,708</td>
<td>2,896</td>
</tr>
<tr>
<td>Incorrect parameter values</td>
<td>11,178</td>
<td>3,955</td>
</tr>
<tr>
<td>Calls to forbidden methods</td>
<td>97 (0.5%)</td>
<td>62</td>
</tr>
<tr>
<td>Insecure compositions</td>
<td>4,443 (21.8%)</td>
<td>1,367</td>
</tr>
<tr>
<td>Total</td>
<td>20,426</td>
<td>4,143</td>
</tr>
</tbody>
</table>

7.2 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 7.1) 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. We detail COGNICRYPT SAST’s findings on apps that do contain misuses in Table 3. 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 in 4,143 apps. As an app may contain multiple errors and, by extension, various types of errors, the total number of apps that contain misuses is not the sum of apps that contain certain misuse types.

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 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 [21]. 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

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%.
insecure algorithm DES or the ECB mode of operation. This result is in line with the findings of prior studies [16, 18, 49].

More than 75% of the typestate errors that CRYPTO 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, CRYPTO 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 CRYPTO 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 CRYPTO 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. Transitivity, CRYPTO SAST reports an unsatisfied predicate also for any Cipher object that relies on the generated key object.

CRYPTO 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 5.

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.

### 7.3 Performance (RQ3)

**Setup**

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

In Section 7.2 we report that CRYPTO SAST found usages of the JCA in 4,349 of all 10,000 apps in our dataset.

Fig. 11. Analysis time (in log scale) of the individual phases of CRYPTO SAST when running on the apps that use the JCA.

If we include in the reporting those usages that arise from misuses within the common libraries previously excluded (see Section 7.2), 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 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, CRYPTO SAST analyzes an app in 101 seconds, with call-graph construction taking most of the time (83%).

### 7.4 Comparison to Existing Tools (RQ4)

**Setup**

We compare CRYPTO SAST to CRYPTO LINT [18], the most closely related tool (see also Section 9.3). Unfortunately, despite contacting the authors we were unable to obtain access to CRYPTO LINT’s implementation. We thus resorted to reimplementing the original rules that are hard-coded in CRYPTO LINT as CrySL rules. All CRYPTO LINT rules can be modelled in CrySL. This rule set, however, still only covers a fraction of what CRYPTO SAST’s default
CrySL rule set covers. This fact alone shows CrySL’s superior expressiveness.

In this section, RULESETFULL denotes this more comprehensive CrySL rule set of CogniCryptSL that we have created for all the JCA classes, while RULESETCL denotes the set of CrySL rules that we developed to model the original CRYPTO-LINT rules. Additionally, CogniCryptSAST denotes our analysis when it runs using RULESETFULL, and CogniCryptCL denotes the analysis when it runs using RULESETCL.

RULESETFULL consists of 23 rules, one for each class of the JCA. RULESETCL comprises only six individual rules, and they only use the sections ENSURES, REQUIRES and CONSTRAINTS. In other words, the original hard-coded CRYPTO-LINT rules do neither comprise typestate properties nor forbidden methods. For three out of six rules, we managed to exactly capture the semantics of the hard-coded CRYPTO-LINT rule in a respective CrySL rule. The remaining three rules (3, 4, and 6 of the original CRYPTO-LINT rules) cannot be perfectly expressed as a CrySL rule, and our CrySL-based rules over-approximate them instead.

CRYPTO-LINT 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 CRYPTO-LINT 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 CRYPTO-LINT rules that we modelled in CrySL. In result, RULESETCL is stricter than the original implementation of CRYPTO-LINT. In the comparison of CogniCryptSAST and CogniCryptCL in terms of their findings, the stricter rules produce more warnings than the original implementation of CRYPTO-LINT. In our comparison against CogniCryptSAST, this setup favours CRYPTO-LINT 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

CogniCryptCL detects usages of JCA classes in 1,866 Android apps. For these apps, CogniCryptCL reports 5,507 misuses, only a third of the 20,426 misuses that CogniCryptSAST identifies using RULESETFULL, our more comprehensive rule set.

Using CogniCryptCL, all reported warnings are related to 6 classes, compared to 23 classes that are specified in RULESETFULL. As we have pointed out, CRYPTO-LINT does not specify any typestate properties or forbidden methods. Hence, CogniCryptCL does not find the 4,805 warnings that CogniCryptSAST identifies in these categories using RULESETFULL. Furthermore, while CogniCryptSAST reports 11,178 constraint violations with the standard rules, CogniCryptCL reports only 1,177 constraint violations. Of the 11,178 constraint violations, 9,958 are due to the rule specification for the class MessageDigest. CRYPTO-LINT does not model this class. If we remove these violations, 1,609 violations are still reported by CogniCryptSAST, a total of 432 more than by CogniCryptCL.

We compare our findings to the study by Egele et al. [18] 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). In contrast, in our study, CogniCryptCL 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. [18]. We see two possible explanations that may contribute to the lower number. First, 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. Second, due to our more extensive rule set, a far greater number of apps actually counts as using cryptography, even those that do not even use Cipher. Hence, the ratio of crypto apps in our findings that even use Cipher is necessarily much smaller than for CRYPTO-LINT’s original evaluation, pushing down the ratio of apps possibly containing this particular misuse.

Based on the high precision (92.6%) and recall (96.2%) values discussed in RQ1, we argue that CogniCryptSAST provides an analysis with a much higher recall than CRYPTO-LINT. Although the larger and more comprehensive rule set, RULESETFULL, detects more complex misuses, the precise analysis keeps the false-positive rate at a low percentage.

7.5 Threats to Validity

Our ruleset RULESETFULL is mainly based on the documentation of the JCA [25]. 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. CogniCryptSAST 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.

8 Crypto-API Misuse in Java Software

In this section, we present a large-scale study of misuses of Crypto APIs in Java applications. With the study, we wish to answer the following research questions:

**RQ5:** How prevalent are misuses of Crypto APIs in Java software?

**RQ6:** What types of misuses are present in Java software?

**RQ7:** How do Java and Android software compare in terms of Crypto APIs misuses?
TABLE 4
Types of API Misuses reported by COGNICRYPT\(_{\text{SAST}}\) for Maven Central artefacts that use the JCA.

<table>
<thead>
<tr>
<th>API Misuse Type</th>
<th># Warnings</th>
<th># Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect calling sequences</td>
<td>8,860 (39.1%)</td>
<td>2,408</td>
</tr>
<tr>
<td>Incorrect parameter values</td>
<td>6,827 (30.1%)</td>
<td>3,656</td>
</tr>
<tr>
<td>Calls to forbidden methods</td>
<td>203 (0.9%)</td>
<td>130</td>
</tr>
<tr>
<td>Insecure compositions</td>
<td>6,774 (29.8%)</td>
<td>1,737</td>
</tr>
<tr>
<td>Total</td>
<td>22,664</td>
<td>7,287</td>
</tr>
</tbody>
</table>

8.1 Setup
To have a representative sample set of Java applications, we collected the latest versions of all artefacts on Maven Central, the world’s largest code repository for Java applications. In May 2018, the index of Maven Central lists a total of 2,768,263 JAR files. We restricted our analysis to the latest version of each individual software artefact, resulting in a dataset of 204,788 JAR files that we ran COGNICRYPT\(_{\text{SAST}}\) on with RULESET\(_{\text{FULL}}\).

We ran the study on a 32-core machine with 128 GB RAM and 2 TB of disk space. We analyzed 10 artefacts at a time in parallel, and granted each analysis a maximum of 10 GB of heap space. Most of the artefacts on Maven Central are libraries, which makes it difficult to pre-compute a call graph [14] for an artefact. We rely on the call graph algorithm Class Hierarchy Analysis (CHA) [17] that constructs an imprecise but efficient call graph that is well suited for libraries. For the artefacts that contain uses of the JCA, it took an arithmetic mean of 38 seconds to construct the call graph and 120 seconds to run COGNICRYPT\(_{\text{SAST}}\) per application. In total, the analysis took 6 days to complete for the whole dataset. To answer RQ6, we compare the results from our study on Maven Central to the study in the previous section.

8.2 Results
Table 4 summarizes the results of the study. COGNICRYPT\(_{\text{SAST}}\) finds 7,288 Java artefacts that use the JCA. Of those, 4,929 artefacts (63.0%) produce at least one warning. In total, these artefacts contain 22,664 misuses, an average of 3.1 misuses per artefact.

**RQ5:** COGNICRYPT\(_{\text{SAST}}\) finds an average of 3.1 misuses per artefact, with at least one misuse in 63% of all artefacts, resulting in an overall lower average than in our Android study.

A more detailed analysis of the results reveals that roughly 39.1% of the misuses are parameter-constraint violations. Similar to our Android study, class MessageDigest is the biggest source of constraint violations (4,462 misuses). The only other class that sticks out is again Cipher with 1,262 misuses. Although we have not manually analyzed a representative number of vulnerability reports from COGNICRYPT\(_{\text{SAST}}\) for this dataset, given the results from our manual analysis in Section 7, we assume most of the misuses related to these two classes come from uses of MD5, SHA-1, DES, and ECB.

**RQ6:** In contrast to our evaluation of Android apps, across all studied Java artefacts on Maven Central, insecure calling sequences (39.1%) contribute the most to the detected misuses, followed by insecure parameters (30.1%).

In Section 7, we concluded that out of the 4,071 apps that contain uses of the JCA, 95% misuse it at least once. Our results indicate that the rate of insecure Java applications is 63% (i.e., 32 percentage points lower). COGNICRYPT\(_{\text{SAST}}\) has also found a lower average of misuses per application for our sample set. For Android, COGNICRYPT\(_{\text{SAST}}\) found 4.8 misuses per app, while here we saw an average of 3.1 misuses per app. Therefore, in terms of overall misuse, Java applications appear to contain fewer misuses, but are still insecure overall. The distribution of misuse types exhibits two remarkable differences. That is, COGNICRYPT\(_{\text{SAST}}\) finds many more applications with incorrect parameters (95.5% vs. 50.1%) and incorrect calling sequences (69.9% vs. 33.0%). For the rest, the numbers are closer to each other. There are more with insecure compositions (33.0% vs. 23.8%) and slightly fewer calls to forbidden methods (1.4% vs. 1.7%).

**RQ7:** Comparing our answers to RQ5 and RQ6 with those to RQ2, we first observe a 34% lower rate of crypto-misusing artefacts in Maven Central than crypto-misusing Android apps in the Google Play Store. The distribution is generally rather similar, only the much lower number of apps with constraint errors is notable.

8.3 Case Studies
We want to take a close look at three vulnerabilities that COGNICRYPT\(_{\text{SAST}}\) detected thanks to its white-list approach and its precise analysis. We encountered these examples when cross-checking some of the findings.

8.3.1 Kerberos Application
We first discuss an example from an artefact implementing the kerberos protocol developed by a widely known vendor. The code snippet in Figure 12 contains two misuses. First, a Cipher object is instantiated for an encryption with the broken algorithm RC4 (Line 127). Second, Line 140 in the method calculateIntegrity() defines a MAC object. This statement is followed by a call to Mac.doFinal(). When executed, this method will throw an IllegalStateException because any MAC object must be initialized by a call to init() before calling doFinal() on it. This misuse not only makes the code non-functional, but also insecure as a security-critical operation, namely mac-ing of data, can never be performed.
139 try {
140     Cipher cipher = Cipher.getInstance("ARCFOUR");
141     SecretKey key = new SecretKeySpec(keyBytes, "ARCFOUR");
142     if (isEncrypt) {
143         cipher.init(Cipher.ENCRYPT_MODE, key);
144     } else {
145         cipher.init(Cipher.DECRYPT_MODE, key);
146     }
147     return cipher.doFinal(data);
148 }
149 }
150
151 public byte[] processCipher(boolean isEncrypt, byte[] data, byte[] keyBytes) {
152     Cipher cipher = Cipher.getInstance("ARCFOUR");
153     SecretKey key = new SecretKeySpec(keyBytes, "ARCFOUR");
154     if (isEncrypt) {
155         cipher.init(Cipher.ENCRYPT_MODE, key);
156     } else {
157         cipher.init(Cipher.DECRYPT_MODE, key);
158     }
159     return cipher.doFinal(data);
160 }
161
162 public byte[] calculateIntegrity(byte[] data, byte[] key, KeyUsage usage) {
163     try {
164         Mac digester = Mac.getInstance("HmacMD5");
165         return digester.doFinal(data);
166     } catch (NoSuchAlgorithmException nsae) {
167         return null;
168     }
169 }
170
171 protected KeyStore getTrustStore() {
172     [...]
173     String truststorePassword = getTrustStorePassword();
174     if ((truststore != null) && (truststorePassword != null)) {
175         ts = getStore(truststoreType, truststore, truststorePassword);
176     }
177     return ts;
178 }
179 }
180
181 protected String getKeystorePassword() {
182     String keyPass = (String)attributes.get("keypass");
183     if (keyPass == null) {
184         keyPass = "changeit";
185     }
186     String keystorePassword = (String)attributes.get("keystorePass");
187     if (keystorePass == null) {
188         keystorePass = keyPass;
189     }
190     return keystorePass;
191 }
192 }
193
194 protected String getTrustStorePassword() {
195     String truststorePassword = (String)attributes.get("truststorePass");
196     if (truststorePassword == null) {
197         truststorePassword = System.getProperty("javax.net.ssl.trustStorePassword");
198     }
199     return truststorePassword;
200 }
201
202 private KeyStore getStore(String type, String path, String pass) {
203     KeyStore ks = KeyStore.getInstance(type);
204     ks.load(istream, pass.toCharArray());
205     return ks;
206 }
207
208 public byte[] processCipher() {
209     byte[] data = "test".getBytes();
210     byte[] keyBytes = new byte[16];
211     Cipher cipher = Cipher.getInstance("ARCFOUR");
212     cipher.init(Cipher.ENCRYPT_MODE, key);
213     byte[] encrypted = cipher.doFinal(data);
214     System.out.println("Encrypted data: " + new String(encrypted));
215     cipher.init(Cipher.DECRYPT_MODE, key);
216     byte[] decrypted = cipher.doFinal(encrypted);
217     System.out.println("Decrypted data: " + new String(decrypted));
218 }
219
220 public byte[] calculateIntegrity() {
221     byte[] data = "test".getBytes();
222     byte[] key = new byte[16];
223     Mac digester = Mac.getInstance("HmacMD5");
224     return digester.doFinal(data);
225 }
226
227 public void main(String[] args) {
228     processCipher();
229     System.out.println("Mac calculated: " + new String(calculateIntegrity()));
230 }
231
8.3.2 Application Server

Fig. 13. A hard-coded password ("changeit"), Line 184, flows to the call to KeyStore.load() in Line 148

Fig. 12. An example illustrating the use of the insecure RC4 and missing the initialization of a MAC object.

8.3.3 Data-Visualization Application

Lastly, we discuss a misuse in the code snippet in Figure 14. As mentioned before, CrySL mostly follows a white-listing approach, except that it also allows for the declaration of forbidden methods. Certain init() methods of class Cipher are examples of those forbidden methods. These init() methods do not allow one to pass IVs or similar extra parameters, which are, however, necessary if one wishes to use a mode of operation other than ECB. Since the proper generation of an IV can be tricky, the standard provider SunJCE can automatically prepare an IV for the developer in case of an encryption. In turn, the developer has to retrieve the IV after the encryption and supply it to the Cipher object responsible for the decryption by calling an appropriate init method. If no IV is provided, the statement throws an InvalidKeyException and is, therefore, not even executed successfully. In summary, should another mode than ECB be used for a decryption with a symmetric block cipher, one must not call Cipher.init() methods that do not take an IV. However, the code snippet in Figure 14 does exactly that.

Lines 184-187 retrieve a secret key, an algorithm, a mode of operation, padding scheme, and an IV from an external context. COGNiCRYPT SAST fails to determine the values precisely, so it considers all possibilities. Line 189 creates a Cipher object configured with the algorithm and
9 RELATED WORK

We now contrast CRYSL and COGNICRYPT, 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.

9.1 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. [11,13] as well as Naem and Lhoták [36] present algorithms to (partially) evaluate state matches prior to program execution, using static analysis.

Martin et al. [32] 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 [32], SQL injection, and cross-site scripting [31]. Compared to tracematches, PQL captures a greater variety of pattern specifications, at the disadvantage of only flow-insensitive static optimizations. PQL serves as the main inspiration for CRYSL’s syntax. Other languages that pursue similar goals include PTQL [23], PDL [34], SLIC [8,9] and TS4J [12].

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 [53], lowering the threat of false warnings.

9.2 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. [46] surveyed over 60 approaches to API property inference. As this survey shows, 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 [16,18,49].

To infer Crypto-API rules, Paletov et al. [41] thus follow a different approach: instead of mining the client code directly, they instead mine code changes related to Crypto APIs. Subsequently, the authors cluster these changes and derive a usage rule from each cluster. While the work is a first step towards inferring Crypto-API rules,
it also shows the challenges involved. For instance, a closer observation of the inferred rules shows that many of them are overly simplistic and lack context. For instance their rule R4 states “SecureRandom with getInstanceStrong should be avoided” although this is only true “on server-side code running on Solaris/Linux/MacOS”—in most other cases, calling getInstanceStrong is actually recommended and avoids security pitfalls. The approach also lacks recall: the paper states 13 rules only, while our rule set for the JCA alone compactly encodes hundreds of individual rules. Nonetheless, it would be interesting to see if the authors’ approach can be used to infer at least partial CrySL rules. For their experiments, Paletov et al. did not automate the actual generation of machine-checkable rules but instead derived appropriate static checks by hand.

Another idea that appears sensible at first sight is to infer correct usage of Crypto APIs from posts on developer portals like StackOverflow. However, recent studies show 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.

9.3 Detecting Misuses of Security APIs

Only few previous approaches specifically address the detection of misuses of security APIs. CrytoLint [15] 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 CrytoLint, because they would have to be hard-coded. Unlike CrytoLint, CrySL is designed to allow crypto experts to also express comprehensive and complex rules with ease. In Section 2, we have extensively compared our tool CogniCryptSL to CrytoLint.

Another tool that finds misuses of Crypto APIs is Crypto Misuse Analyzer (CMA) [19]. Similar to CrytoLint, CMA’s rules are hard-coded, and its static analysis is rather basic. Many of CMA’s hard-coded rules are also contained in the CrySL rule set that we provide. Unlike CogniCryptSL, CMA has been evaluated on a small dataset of only 45 apps.

Chatzikonstantinou et al. [16] 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.

Nguyen et al. [38] present FixDroid. The static-analysis plugin for Android Studio comes equipped with 13 rules related to security APIs. In terms of Crypto APIs, it also covers about the same rules as CrytoLint.

Wang et al. [58] present NativeSpeaker, a tool that checks for crypto misuses in native code. The tool can detect two kinds of crypto uses. First, it detects when native code calls the JCA (whose interfaces are implemented in plain Java). Second, it applies heuristics comprising filters on an operation’s type and name to find cryptography within the native code itself. For each use found, it checks for a number of misuse types related to symmetric encryption only. In this context, NativeSpeaker finds uses of outdated crypto algorithms, uses of ECB mode, and improper key material.

Braga et al. [14] present a comparative survey of free static analyzers that check for misuses of crypto APIs. The studied tools include FindSecBugs [5], VisualCodeGrepper [57], Xanitizer [45], sonar-scanner [50], and Yasca [48]. To evaluate these tools, the authors compile a benchmark of 384 test cases, 202 of which contain crypto misuses. When applying each tool to their benchmark, they find the general coverage of crypto misuses to be rather low. Xanitizer – the best among the selected – only finds 68 misuses while producing 40 false positives. The tools mostly cover simple misuses such as outdated algorithms or ECB mode, but fail on more complex cases like detecting improper IVs.

Other work has investigated other kinds of security APIs. Fahl et al. [19] analyzed 13,500 Android apps with their static checker Malldroid. Malldroid evaluates apps in terms of insufficient validation of TLS certificates. From their sample set, 1,074 apps do prove to fall short in that regard, leaving them vulnerable to person-in-the-middle attacks. Similarly, Georgiev et al. [22] achieve similar results in an in-depth analysis of how a number of high-profile apps handle TLS-certificate validation.

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’s code, making it hard for non-experts in static analysis to extend or alter the rule set. Consequently, the tools are not completely incapable of supporting CogniCryptSL’s broad range of misuses, but extending one to do so requires intrinsic knowledge of the respective tool and its code. This limitation also makes it 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 without expertise in static analysis to 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.

10 CONCLUSION

In this paper, we present CrySL, a specification 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 CogniCryptSL checking for compliance according to the rules. Applying CogniCryptSL, the analysis for our extensive ruleset RULESET_FULL to 10,000 Android apps, we found 20,426 misuses spread over 95% of the 4,349
apps using the JCA. Similarly, we applied COGNICryptSAST to 2,700,000 artefacts on Maven and it detected misuses in 63% of the artefacts that use cryptography. COGNICryptSAST is also highly efficient: it analyzed all of Maven Central in under a week and for more than 75% of the apps the analysis finishes in under 3 minutes, where most of the time is spent on call graph construction.

11 Future Work

In future work, we plan to address the following challenges. 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 COGNICryptSAST. 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 also intend on applying CRYSL in other contexts. One of the authors of this paper has some students implementing 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 at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle (39) are being used as well and we will extend COGNICryptSAST to support them. Additionally, we will investigate to what extent CRYSL is applicable to Crypto APIs in other programming languages. At the time of writing, we are exploring CRYSL’s compatibility with OpenSSL (40). We finally aim to examine whether CRYSL is expressive enough to meaningfully specify usage constraints for non-crypto APIs.

Lastly, we hope that in the future, domain experts model their own cryptographic libraries in CRYSL, such that developers using the libraries benefit from the static analysis support offered by COGNICrypt.

Acknowledgements

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, the Heinz Nixdorf Foundation, a Fraunhofer ATTRACT grant, and 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. We finally thank Andreas Dann, Sarah Nadi, Michael Reif, Lisa Nguyen Quang Do, and Michael Eichberg for their help with and early feedback on CRYSL.

References


[18] Manuel Egele, David Brumley, Yanick Fratantonio, and Jeffrey Dean. Opti-


[45] RigsIT. Xanitizer.


Stefan Krüger is a PhD Student at Paderborn University and a member of the collaborative research center CROSSING. Krüger's main research interests are API usability, DSLs for the specification of security properties of programs, and automated detection of crypto API misuses.

Johannes Späth is a Research Associate at the Software Engineering and IT Security Department of Fraunhofer IEM in Paderborn, Germany. His research focuses on static analysis where he enjoys developing efficient and precise algorithms, e.g., points-to or typestate analysis.

Karim Ali is currently an Assistant Professor in the Department of Computing Science at the University of Alberta. His research interests are in programming languages and software engineering, particularly in scalability, precision, and usability of program analysis tools.

Eric Bodden is a full professor for Secure Software Engineering at the Heinz Nixdorf Institute of Paderborn University, Germany. He is further the director for Software Engineering at the Fraunhofer Institute for Engineering Mechatronic Systems.
Mira Mezini is a full professor for Software Technology at TU Darmstadt, Germany. In her main research, she links programming languages, software security, and program analysis.