# CogniCrypt— Sichere Integration Kryptographischer Software[1]

Stefan Krüger[2]

**Abstract:** Empirische Studien zeigen, dass Fehlbenutzungen von Crypto APIs weit verbreitet sind. Die Literatur liefert mehrere Ansätze, diese zu beheben, aber keiner adressiert das Problem vollständig. Das Resultat ist eine lückenhafte Landschaft verschiedener Ansätze. In dieser Arbeit adressiere ich das Problem solcher Fehlbenutzungen systematisch durch CogniCrypt. CogniCrypt integriert verschiedene Arten von Tool Support in einen Ansatz, der Entwickler davon befreit, wissen zu müssen, wie die APIs benutzt werden. Zentral für meinen Ansatz ist CrySL, eine Beschreibungssprache, mit der spezifiziert wird, wie APIs benutzt werden. Ich habe einen Compiler für CrySL und zwei darauf aufsetzende Supporttools, die Code-Analyse CogniCrypt$_{SAST}$ und den Code-Generator CogniCrypt$_{GEN}$, entwickelt. Schlussendlich habe ich CogniCrypt prototypisch implementiert und in einer empirischen Evaluierung die Effektivität von CogniCrypt gezeigt.

## 1 Einleitung

Digitale Geräte werden vielfach verwendet um sensitive Daten zu speichern. Kryptographie ist das Hauptwerkzeug um solche Daten vor Fälschung zu schützen. Damit dieser Schutz effektiv ist, müssen Algorithmen nicht nur konzeptionell sicher und korrekt implementiert sein, sondern auch korrekt in den Anwendungscode integriert werden. Das scheint jedoch oft nicht zu funktionieren. Lazar et al. [La14] untersuchten 269 kryptographische Schwachstellen und stellten fest, dass 83% dieser von Entwicklern verursacht werden, die Algorithmen unsicher integrieren. Nachfolgende Forschung zu diesem Phänomen kam zu dem Schluss, dass das Problem von signifikanter Größe ist [Eg13, Gu19, Fe19, Ve17]. Die Gründe hinter jedem individuellen Fehler mögen mannigfaltig sein. Das Design der kryptographischen Bibliotheken und ihrer Anwendungsschnittstellen (API) sind aber wohl zentral für die Schwierigkeiten von Entwicklern. Das zeigt auch folgendes Beispiel.

### 1.1 Ein Motivierendes Beispiel

Die Java Cryptography Architecture (JCA) [Or20], Javas Haupt-Crypto-API, stellt die API `PBEKeySpec` zur passwort-basierten Generierung von Schlüsseln bereit. Abbildung 1 zeigt eine Nutzung von `PBEKeySpec` wie sie in unzähligen realen Programmen zu finden ist.

Das Codebeispiel erstellt zunächst ein `PBEKeySpec`-Objekt durch den Aufruf des Konstruktors. Dieser erwartet ein Passwort, einen Salt, eine Iterationszahl und die Schlüsselänge.

[1] Englischer Originaltitel der Dissertation: "CogniCrypt– The Secure Integration of Cryptographic Software"

[2] krueger@cqse.eu

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

Abb. 1: Beispiel einer *inkorrekten* passwortbasierten Schlüsselgenerierung in Java.

Das `PBEKeySpec`-Objekt wird an ein `SecretKeyFactory`-Objekt weitergereicht, das den Schlüsselableitungsalgorithmus `PBKDF2` verwendet um Schlüsselmaterial zu generieren (Zeilen 5–6). In der letzten Zeile erstellt das Code-Beispiel daraus dann den Schlüssel.

Das Beispiel enthält mehrere Fehler um den Konstruktor-Aufruf von `PBEKeySpec`. Der erste liegt beim Salt. Dieser muss zufällig und unvorhersehrbar sein [Bu17], im Codebeispiel ist er aber festverdrahtet. Der zweite Fehler betrifft das Passwort. Der Konstruktor erwartet das Passwort aus gutem Grund als Char-Array: Passwörter sollten nicht länger im Speicher bleiben als notwendig, das heißt, sie sollten aufgeräumt werden. Strings sind allerdings unveränderbar, sie aufzuräumen ist unmöglich. Zuletzt fehlt im Code ein Aufruf der Methode `clearPassword()`. Trotz der Fehler läuft der Code ohne Exceptions.

### 1.2 Beiträge Dieser Arbeit

Die Probleme im gezeigten Code-Beispiel mögen wie Einzelfälle aussehen, wie die obige Forschung aber zeigt, sind sie es aber nicht. Eine Vielzahl unterschiedlicher Ansätze hat versucht das Problem anzugehen. Neben den oben referenzierten Studien und deren Programmanalysen wurde unter Anderem versucht, fehlerhafte Programme zu reparieren, Hilfsmittel für APIs zu erweitern oder bessere Designs für APIs zu entwickeln. Diese Ansätze sind hilfreich: Sie fördern das Verständnis über gute API-Nutzbarkeit, sie verringern bestehende Probleme oder können den Weg ebnen für neue APIs. Am Ende liefern sie allerdings keine systematische Lösung. Meine Arbeit geht daher einen anderen Weg.

Man sieht, dass Anwendungsentwickler einerseits sofortige Hilfe benötigen. Daher unterstützt meine Lösung Entwickler, wenn sie die APIs nutzen. Es ist weiterhin klar, dass Unterstützung aus verschiedenen Richtungen kommen muss. Nur eine Programmanalyse oder nur bessere Hilfsmittel genügen nicht. Stattdessen präsentiere ich eine Lösung, die mehrere Formen von Unterstützung integriert. Meine Arbeit hat insgesamt folgendes Ziel:

*Fehlbenutzungen von Crypto APIs können signifikant reduziert werden durch einen integrierten Ansatz, der Entwickler davon befreit, wissen zu müssen, wie die APIs genutzt werden müssen.*

Zur Realisierung dieses Ziels präsentiere ich den integrierten Ansatz CogniCrypt. Ich habe CogniCrypt als Plugin für die IDE Eclipse entworfen und prototypisch implementiert, damit es sich in den Arbeitsablauf von Entwicklern integriert. CogniCrypt besteht

aus mehreren Komponenten. Dem Ansatz zugrunde liegt die textuelle Spezifikationssprache CRYSL (Abschnitt 3). Mit Hilfe von CRYSL können Domänen-Experten definieren, wie eine API benutzt werden soll. Ich habe die JCA in CRYSL modelliert. Auf Basis von CRYSL können verschiedene Formen von Tool-Unterstützung gebaut werden.

Eine Form von Tool Support liefert COGNICRYPT$_{SAST}$ (Abschnitt 4), eine statische Analyse, die ein Java- oder Android-Programm auf ihre Befolgung von CRYSL-Regeln hin überprüft. Ich habe weiterhin den Code-Generator COGNICRYPT$_{GEN}$ (Abschnitt 5) entwickelt. Dieser generiert anwendungsfallspezifischen sicheren Code für Crypto APIs.

Zuletzt habe ich die Effektivität von COGNICRYPT in einem kontrollierten Experiment evaluiert (Abschnitt 6). Die Ergebnisse zeigen, dass Entiwckler mit COGNICRYPT sowohl sichereren als auch funktionaleren Code produzieren. Ausführlich habe ich meine Arbeit und ihre Ergebnisse in meiner Dissertation [Kr20] festgehalten.

## 2 COGNICRYPT

Ich präsentiere COGNICRYPT hier zunächst aus einer Nutzerperspektive um zu demonstrieren, wie es Entwickler unterstützt. Meine prototypische Implementierung für Eclipse ist öffentlich verfügbar und open-source[3]. Aktuell unterstützt der Prototyp nur Java, die vorgestellten Konzepte sind allerdings nicht auf Java beschränkt. Das Eclipse-Plugin COGNICRYPT kombiniert die beiden Ansätze COGNICRYPT$_{GEN}$ und COGNICRYPT$_{SAST}$.

Der Code-Generator COGNICRYPT$_{GEN}$ kann *sicheren* Code für unter anderem das Beispiel in Abbildung 1 generieren. Dazu klickt der Nutzer den COGNICRYPT$_{GEN}$-Button in der Toolbar von Eclipse und wählt im erscheinenden Dialogfenster den passenden Anwendungsfall aus. Anschließend beantwortet er mehrere Fragen, die COGNICRYPT$_{GEN}$ zur Konfiguration verwendet. Zuletzt wählt er die Klasse aus, in die COGNICRYPT Code generiert. Im Anschluss generiert COGNICRYPT$_{GEN}$ zwei Artefakte. Einmal generiert es die eigentliche Implementierung für den ausgewählten Anwendungsfall in das Paket `de.cognicrypt.crypto`. Dazu generiert COGNICRYPT eine Methode `templateUsage()`, die zeigt, wie die Implementierung aufgerufen wird, in die vorher ausgewählte Klasse.

COGNICRYPT benachrichtigt Nutzer über Fehlbenutzungen von Crypto APIs, indem es kontinuierlich die Analyse COGNICRYPT$_{SAST}$ laufen lässt. COGNICRYPT$_{SAST}$ stellt sicher, dass die API-Nutzungen sicher bleiben, wenn Entwickler sie ändern. Darüber hinaus hilft COGNICRYPT$_{SAST}$ auch, wenn Entwickler APIs direkt, also ohne COGNICRYPT$_{GEN}$, nutzen. Bei einer Fehlbenutzung generiert COGNICRYPT einen Eclipse Error Marker.

## 3 CrySL

Damit COGNICRYPT weiß, wie Crypto APIs verwendet werden, benutzt es Regeln in CRYSL. CRYSL ist eine Spezifikationssprache, die Experten in die Lage versetzt zu de-

[3] `www.cognicrypt.org`

finieren, wie eine API genutzt wird. Die Sprache folgt größtenteils einem White-Listing-Ansatz und die Syntax ist bewusst einfach und Java-nah gehalten.

### 3.1 Ausdruckskraft von CrySL

Eine Regel besteht aus bis zu vier Abschnitten: (1) Ein Methoden-Sequenz-Muster, (2) Nutzungsauflagen für Methodenparameter, (3) Verbotene Methoden und (4) Nutzungsauflagen für Interaktionen verschiedener Klassen.

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

Abb. 2: CrySL-Regel für JCA-Klasse `javax.crypto.spec.PBEKeySpec`.

Zur Illustration von CrySL zeige ich in Abbildung 2 die CrySL-Regel von `PBEKeySpec`. Die Regel definiert unter dem Schlüsselwort `SPEC`, welche Klasse sie spezifiziert. Unter `OBJECTS` listet sie dann die vier in der Regel verwendeten Objekte. Eines davon ist das `char[]`-Objekt `password`. Die Abschnitte `EVENTS` und `ORDER` spezifizieren das Methoden-Sequenz-Muster. Dazu werden unter `EVENTS` alle Methoden, die zur korrekten Verwendung von `PBEKeySpec` beitragen können, als Methoden-Event-Muster definiert (Zeilen 18–19). Der `ORDER`-Abschnitt definiert anschließend valide Aufrufsequenzen der Methoden-Events als regulären Ausdruck. Mit Hilfe von Labels (z.B. `cP`) vereinfacht CrySL die spätere Referenz auf die Methoden. Das Muster für `PBEKeySpec` ist relativ einfach: Der Konstruktor `c1` muss einmal aufgerufen werden, gefolgt von einem Aufruf von `cP`. Der Abschnitt `CONSTRAINTS` fügt die oben erwähnten Nutzungsauflagen für Parameter hinzu.

Zur Definition von Auflagen für Interaktionen von Klassen bietet CrySL drei weitere Schlüsselwörter an, die einen Rely/Guarantee-Mechanismus implementieren. Zuerst ist da `ENSURES`, mit dem eine Klasse eine Garantie ausgibt, wenn sie richtig verwendet wurde. `PBEKeySpec` garantiert `speccedKey`, also dass ein korrekt instantiiertes `PBEKeySpec`-

Objekt sicheres Schlüsselmaterial bereitstellt. Die Regel nutzt das Schlüsselwort `after` um anzuzeigen, dass die Garantie nach dem Aufruf des Konstruktors gegeben ist. Über das Schlüsselwort `NEGATES` zerstört `PBEKeySpec` seine `speccedKey`-Garantie nach dem Aufruf von `cP` wieder. Eine Garantie zu verlangen geschieht über das Schlüsselwort `REQUIRES`. `PBEKeySpec` selbst verlangt, dass sein Salt von einer anderen Klasse randomisiert wurde. Ähnlich können auch Regeln die Garantie `speccedKey` verlangen. Insgesamt deckt die CrySL-Regel von `PBEKeySpec` alle oben diskutierten Fehlbenutzungen ab.

### 3.2 Implementierung

Ich habe die gesamte JCA in insgesamt 23 CrySL-Regeln modelliert. Darüber hinaus wurden unabhängig von dieser Arbeit noch die APIs Google Tink, Bouncy Castle und Bouncy Castle als JCA-Provider in CrySL modelliert. Weiterhin habe ich einen auf dem Sprachentwicklungswerkzeug XText basierenden Compiler für CrySL implementiert. Xtext stellt auf Basis der Grammatik von CrySL grundlegenden Support wie einen Editor bereit. Ich habe auch einen Parser entwickelt, der CrySL-Regeln in ein Java-Objekt-Modell übersetzt, das von auf CrySL aufbauendem Tool-Support verwendet werden kann und von CogniCrypt$_{SAST}$ und CogniCrypt$_{GEN}$ verwendet wird.

## 4 CogniCrypt$_{SAST}$

CogniCrypt$_{SAST}$ ist eine auf CrySL aufsetzende statische fluss- und kontextsensitive Datenflussanalyse. Als Eingabe erhält CogniCrypt$_{SAST}$ neben dem zu analysierenden Java- oder Android-Programm eine Menge von CrySL-Regeln. Im Folgenden analysiert CogniCrypt$_{SAST}$ dann, ob das Programm die CrySL-Regeln einhält.

### 4.1 Ablauf & Prototypische Implementierung

Zunächst parst CogniCrypt$_{SAST}$ die Regeln mit dem CrySL-Compiler. Anschließend überführt CogniCrypt$_{SAST}$ das Zielprogramm mit Hilfe des Analyseframeworks Soot in die Zwischenrepräsentation Jimple und erstellt einen Call Graph vom Zielprogramm. Den Call Graph durchsucht CogniCrypt$_{SAST}$ im Anschluss nach verbotenen Methoden.

Im Anschluss ermittelt CogniCrypt$_{SAST}$ in einer Prä-Analyse die zu analysierenden Objekte, wie das `PBEKeySpec`-Objekt in Abbildung 1. Objekte werden hier über Allocation Sites approximiert. Dabei zeichnet es ebenfalls die Methodenaufrufe auf den Objekten und die potentiellen Laufzeitwerte deren Parameter auf. Ich habe einen Constraint Solver entwickelt, der diese Werte mit jenen in der entsprechenden CrySL-Regel vergleicht. Für `PBEKeySpec` in Abbildung 1 gilt das beispielsweise für die Nutzungsauflage für `iterationCount`. Um die Einhaltung der Methoden-Sequenz-Muster zu überprüfen, übersetzt CogniCrypt$_{SAST}$ diese in Zustandsautomaten und führt mit diesem und den

aufgezeichneten Methodenaufrufen über das Datenflussanalyseframework IDE$^{al}$ eine Typestate-Analyse durch. Für `PBEKeySpec` in Abbildung 1 findet CogniCrypt$_{SAST}$ einen Typestate-Fehler, da es `clearPassword()` nicht aufruft, um das Passwort aufzuräumen.

Es gibt auch für die Interaktion von Klassen eine eigene Analyse in CogniCrypt$_{SAST}$, die dem in Abschnitt 3 beschriebenen Ablauf folgt. Für die richtige Auflösung der Garantien und Versicherungen, hält CogniCrypt$_{SAST}$ während der Analyse eine Liste von Garantien vor. Wenn CogniCrypt$_{SAST}$ beispielsweise das `PBEKeySpec`-Objekt in Abbildung 1 analysiert, überprüft es, ob die Garantie vorliegt, dass `salt` randomisiert ist. Wenn das der Fall ist und das `PBEKeySpec`-Objekt keine weiteren Teile der Regeln verletzen würde, generierte CogniCrypt$_{SAST}$ auch dessen Garantie. Da das Objekt jedoch, wie oben dargelegt, mehrfach falsch benutzt wird, stellt CogniCrypt$_{SAST}$ keine Garantie aus.

### 4.2 Evaluierung

Für die Evaluierung von CogniCrypt$_{SAST}$ habe ich diese Forschungsfragen betrachtet:

**RQ1** Wie hoch sind Präzision und Recall von CogniCrypt$_{SAST}$?
**RQ2** Welche Arten von Fehlbenutzungen findet CogniCrypt$_{SAST}$?
**RQ3** Wie schnell läuft CogniCrypt$_{SAST}$?
**RQ4** Wie gut schneidet CogniCrypt$_{SAST}$ ab im Vergleich mit dem Stand der Technik?

#### 4.2.1 Setup

Zur Beantwortung habe ich CogniCrypt$_{SAST}$ mit dem JCA-Regelsatz auf 10.000 Android-Apps angewendet. Zur Beantwortung von **RQ1** habe ich 50 der Apps zufällig ausgewählt und die Typestate- und Parameterergebnisse von CogniCrypt$_{SAST}$ manuell verifiziert. Um **RQ3** zu beantworten, habe ich die Analysedauer für alle Apps gemessen. Für **RQ4** habe ich CryptoLint [Eg13] auch auf die 10,000 Apps angewendet. Ich hatte keinen Zugriff auf das Tool, sondern haben das Tool simuliert, indem ich CogniCrypt$_{SAST}$ mit CrySL-Regeln, die denen von CryptoLint entsprachen, laufen ließ.

#### 4.2.2 Ergebnisse

**RQ1**: CogniCrypt$_{SAST}$ findet insgesamt 156 Fehlbenutzungen in den 50 Apps. Von den laut CogniCrypt$_{SAST}$ insgesamt 27 Typestate-Fehlern konnte ich 25 bestätigen. CogniCrypt$_{SAST}$ übersieht aufgrund der verwendeten Alias-Analyse aber vier Typestate-Fehler. CogniCrypt$_{SAST}$ findet 129 Fehler bei Nutzungsauflagen für Parameter, 19 davon sind falsch positiv. Ich konnte keine Falschnegativen finden. Insgesamt ergeben sich daher für die Typestate-Analyse 92.6% Präzision und 86.2% Recall, für die Analyse der Nutzungsauflagen fü Parameter dagegen 85.3% Präzision und ein Recall von 100%.

**RQ2**: CogniCrypt$_{SAST}$ findet in 4.439 Apps JCA-Nutzungen, in 95% davon mindestens einen Fehler. Mit 3.955 Apps treten Fehler bei den Nutzungsauflagen für Parameter am häufigsten auf, gefolgt von 2.896 Apps mit Typestate-Fehlern. Unsichere Interaktionen von Klassen kommen in 1.367 Apps vor, Aufrufe verbotener Methoden nur in 62 Apps.

**RQ3**: Im Durchschnitt dauert die Analyse einer App 101 Sekunden, die Zeiten schwanken zwischen 10 Sekunden und 28,6 Minuten. Die Ursache dafür liegt in den Größen der Apps, da CogniCrypt$_{SAST}$ 83% der Analysezeit auf die Erstellung des Call-Graphs verwendet.

**RQ4**: CogniCrypt$_{SAST}$ entdeckt 20.426 Fehlbenutzungen über 23 Klassen verteilt in den 10.000 Android-Apps. Im Kontrast dazu findet das nachgebaute CryptoLint nur 5.507 in sechs Klassen. Insgesamt findet CogniCrypt$_{SAST}$ also fast vier mal so viele Fehler.

## 5 CogniCrypt$_{GEN}$

CogniCrypt$_{GEN}$ ist ein Code-Generator für Crypto APIs, der auf CrySL aufsetzt. CogniCrypt$_{GEN}$ generiert anwendungsfallspezifischen Code, der sicher und korrekt in Abhängigkeit der CogniCrypt$_{GEN}$ bereitgestellten CrySL-Regeln ist. Als weitere Eingabe neben den CrySL-Regeln erhält CogniCrypt$_{GEN}$ ein Code-Template, das sicherheitsunkritischen Wrapper-Code für den jeweiligen Anwendungsfall bereitstellt.

### 5.1 Ablauf & Prototypische Implementierung

```
34  byte[] salt = new byte[32];
35  char[] pwd = {'p', 'w', 'd'}; //Festverdrahtet zur Demonstration
36  javax.crypto.SecretKey encryptionKey = null;
37
38  CrySLCodeGenerator.getInstance().
39   includeClass("java.security.SecureRandom").addParameter(salt,"salt").
40   includeClass("java.security.PBEKeySpec").addParameter(pwd,"password").
41   includeClass("javax.crypto.SecretKeyFactory").
42   includeClass("java.security.SecretKey").
43   includeClass("javax.crypto.SecretKeySpec").addReturnObject(encryptionKey).
44   generate();
```

Abb. 3: CogniCrypt$_{GEN}$-Template, das korrekte Variante von Abbildung 1 generiert.

Um zu illustrieren, wie CogniCrypt$_{GEN}$ funktioniert, beziehe ich mich auf das Template in Abbildung 3, mit dessen Hilfe CogniCrypt$_{GEN}$ eine *sichere* Variante des `PBEKeySpec`-Beispiels in Abschnitt 1 generiert. Zeilen 34 – 36 erstellen drei Objekte und bilden den Wrapper-Teil des Templates. Im Anschluss folgt der Aufruf des eigentlichen Code-Generators durch eine Fluent API. Zwischen Instantiierung und Abschluss (Zeilen 38 und 44) erfolgt dessen Konfiguration über drei Methoden. Durch den Aufruf von `includeClass()` in Zeile 40 generiert CogniCrypt$_{GEN}$ Code für die Klasse `PBEKeySpec`. Der Aufruf von `addParameter()` gibt dann die Variable `pwd` im Wrapper-Code, also das nutzerspezifizierte Passwort, über die Variable `password` in der CrySL-Regel von `PBEKeySpec` an den generierten Code weiter. Die Rückgabe von Objekten vom generierten an den Wrapper-Code erfolgt über die Methode `addReturn()`, wie in Zeile 43 zu sehen. Hier wird der ge-

nerierte Schlüssel in der Code-Template-Variable `encryptionKey` gespeichert. Nach der Verarbeitung des Templates, generiert CogniCrypt$_{GEN}$ die `templateUsage()`-Methode.

Die Implementierung von CogniCrypt$_{GEN}$ setzt auf existierender Infrastruktur auf. So nutzt CogniCrypt$_{GEN}$ den CrySL-Compiler um die Regeln einzulesen. Die Lösung zur Verarbeitung der Templates und Generierung des Codes setzt auf dem Eclipse JDT auf.

## 5.2 Evaluierung

In der Evaluation von CogniCrypt$_{GEN}$ habe ich folgende Forschungsfragen betrachtet:

**RQ5** Unterstützt CogniCrypt$_{GEN}$ oft genutzte kryptographische Anwendungsfälle?
**RQ6** Produziert CogniCrypt$_{GEN}$ schnell genug Code um in der alltäglichen Softwareentwicklung eingesetzt werden zu können?
**RQ7** Nehmen Entwickler von CogniCrypt$_{GEN}$ das Tool als nutzbarer wahr als einen vergleichbaren Code-Generator?

### 5.2.1 Setup

Zur Beantwortung von **RQ5** habe ich aus verschiedenen Quellen Anwendungsfälle zusammengetragen und diese mit CogniCrypt$_{GEN}$ implementiert. Um **RQ6** zu beantworten, habe ich anschließend alle implementierten Anwendungsfälle mit CogniCrypt$_{GEN}$ generiert und die Laufzeit gemessen. Für die Beantwortung von **RQ7** habe ich 16 Doktoranden und Studierende an meiner lokalen Universität gebeten Implementierungen mehrerer Anwendungsfälle in CogniCrypt$_{GEN}$ und einem ähnlichen Code-Generator, der auf XSL basiert, vorzunehmen und anschließend ihre Eindrücke zu schildern.

### 5.2.2 Ergebnisse

**RQ5**: Insgesamt habe ich elf Anwendungsfälle aus drei Quellen zusammengetragen. Die Anwendungsfälle reichen von passwort-basierter Verschlüsselung von Dateien und hybrider Verschlüsselung von Byte-Arrays bis zum Signieren von Strings. Ich habe alle elf Anwendungsfälle erfolgreich in CogniCrypt$_{GEN}$ implementiert.

**RQ6**: Die Laufzeit von CogniCrypt$_{GEN}$ schwankte für die elf Anwendungsfälle zwischen 6,6 und 8,1 Sekunden. CogniCrypt$_{GEN}$ ist also ausreichend performant.

**RQ7**: Alle 16 Teilnehmer schlossen die an sie gestellten Aufgaben erfolgreich ab. Wenn gebeten die Nutzbarkeit der beiden Code-Generatoren zu vergleichen, bewerteten Teilnehmer CogniCrypt$_{GEN}$ als signifikant nutzbarer als die XSL-basierte Lösung.

# 6 Nutzerstudie

Ich evaluiere schließlich die Effektivität von CogniCrypt und somit ob es das oben aufgestellte Ziel erfüllt. Für genauere Beobachtungen betrachte ich diese Forschungsfragen:

**RQ8** Welchen Einfluss hat CogniCrypt auf die Funktionalität der Anwendungen?
**RQ9** Welchen Einfluss hat CogniCrypt auf die Sicherheit der Anwendungen?
**RQ10** Welchen Einfluss hat CogniCrypt auf die Entwicklungszeit der Anwendungen?
**RQ11** Nehmen Entwickler CogniCrypt als nutzbarer wahr im Vergleich zu Eclipse?

## 6.1 Setup

Zur Beantwortung der vier Forschungsfragen führe ich eine Nutzerstudie an der prototypischen Eclipse-Implementierung durch. Dazu habe ich 24 Masterstudierende rekrutiert und diese zwei kryptographische Programme implementieren lassen, eins mit CogniCrypt, eins mit einem regulären Eclipse. Für **RQ8** und **RQ9** habe ich die Lösungen der Teilnehmer mit vorher definierten Sicherheits- und Funktionalitätskriterien abgeglichen. Um **RQ10** zu beantworten, habe ich die Zeiten gemessen, die Teilnehmer für ihre Lösungen brauchten. Für **RQ11** habe ich schließlich alle Teilnehmer zu ihren Eindrücken interviewt.

## 6.2 Ergebnisse

**RQ8**: Insgesamt waren die mit CogniCrypt erstellten Lösungen signifikant funktionaler. Von den 22 mit CogniCrypt implementierten Programme wurden 18 komplett richtig implementiert. Von den ohne CogniCrypt entwickelten hingegen waren nur drei korrekt.

**RQ9**: Die Sicherheitsanalyse liefert nochmal eindeutigere Ergebnisse. Ohne CogniCrypt implementierte nur ein Teilnehmer eine sichere Lösung. Mit CogniCrypt waren dagegen 18 von 22 Programme sicher. Bei einer der Aufgaben waren alle Programme mit CogniCrypt sicher, bei der anderen erhielten waren sie zu 87.5% sicher.

**RQ10**: Für eine der Aufgaben existieren signifikant weniger funktionale Lösungen ohne CogniCrypt als mit CogniCrypt. Daher vergleiche ich die Entwicklungszeit nur für die andere Aufgabe. Ohne CogniCrypt haben Teilnehmer im Schnitt zwischen 16 und 27,5 Minuten benötigt, mit CogniCrypt waren es nur zwischen neun und 20 Minuten. Insgesamt waren Teilnehmer mit CogniCrypt also signifikant schneller.

**RQ11**: Teilnehmer bewerten CogniCrypt generell positiv und signifikant besser als das reguläre Eclipse. Eclipse wurde als schlecht nutzbar bewertet. Einige Teilnehmer hatten aber auch Kritik an CogniCrypt, insbesondere in Bezug auf die Integration in Eclipse.

# 7 Schlussfolgerungen

In dieser Arbeit habe ich den integrierten Ansatz CogniCrypt vorgestellt. Wie ich gezeigt habe, reduziert CogniCrypt signifikant Fehlbenutzungen kryptographischer APIs

durch eine Mischung aus Code-Analyse und Code-Generierung. Diese ermöglicht es Entwicklern Crypto APIs zu nutzen, ohne sich in sie einarbeiten zu müssen.

Durch seinen modularen Aufbau mit der Spezifikationssprache CrySL als Basis für verschiedene Formen von Toolsupport ist CogniCrypt vielfach erweiterbar. Abseits dieser Arbeit wurden bereits das Program-Repair-Tool CogniCrypt$_{\text{FIX}}$, der Testfallgenerator CogniCrypt$_{\text{TEST}}$, der Dokumentationsgenerator CogniCrypt$_{\text{DOC}}$, ein auf OpenJ9-basierendes Hot-Patching-System und eine JCA-Erweiterung, die mit AspectJ Fehlbenutzungen automatisch umgeht, entwickelt. Das zeigt die Wirkmacht con CrySL und der hier vorgestellten Lösung. Andere Erweiterungen auf Basis von CrySL sind denkbar.

## Literaturverzeichnis

[Bu17] Bundesamt fuer Sicherheit in der Informationstechnik (BSI): Cryptographic Mechanisms: Recommendations and Key Lengths. Bericht BSI TR-02102-1, BSI, Januar 2017.

[Eg13] Egele, Manuel; Brumley, David; Fratantonio, Yanick; Kruegel, Christopher: An empirical study of cryptographic misuse in android applications. In: ACM Conference on Computer and Communications Security. S. 73–84, 2013.

[Fe19] Feichtner, Johannes: A Comparative Study of Misapplied Crypto in Android and iOS Applications. In: Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019. S. 96–108, 2019.

[Gu19] Gu, Zuxing; Wu, Jiecheng; Liu, Jiaxiang; Zhou, Min; Gu, Ming: An Empirical Study on API-Misuse Bugs in Open-Source C Programs. In: 43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1. S. 11–20, 2019.

[Kr20] Krüger, Stefan: CogniCrypt - The Secure Integration of Cryptographic Software. Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Softwaretechnik, Oktober 2020.

[La14] Lazar, David; Chen, Haogang; Wang, Xi; Zeldovich, Nickolai: Why does cryptographic software fail?: a case study and open problems. In: ACM Asia-Pacific Workshop on Systems (APSys). S. 7:1–7:7, 2014.

[Or20] Oracle Inc.: , Java Cryptography Architecture (JCA), 2020. `https://docs.oracle.com/en/java/javase/15/security/java-cryptography-architecture-jca-reference-guide.html`.

[Ve17] VeraCode: , State of Software Security 2017. `https://info.veracode.com/report-state-of-software-security.html`, 2017.

**Stefan Krüger** hat von 2009 bis 2014 an der Otto-von-Guericke Universität Magdeburg Informatik studiert und mit dem Master of Science abgeschlossen. Von 2015 an hat Krüger zunächst an der TU Darmstadt, dann ab April 2016 an der Universität Paderborn zur sicheren Integration kryptographischer Software promoviert. Krügers Arbeit fand im Rahmen des Sonderforschungsbereich CROSSING statt. Dessen Ziel ist es Lösungen für zukunftssichere Kryptographie und IT-Sicherheit zu entwickeln. Inzwischen arbeitet Krüger für die CQSE GmbH.