Runtime Verification of Crypto APIs: An Empirical Study

Adriano Torres, Pedro Costa, Luis Amaral, Jonata Pastro, Rodrigo Bonifácio, Marcelo d'Amorim, Owolabi Legunsen, Eric Bodden, and Edna Dias Canedo

***Abstract*—Misuse of cryptographic (crypto) APIs is a noteworthy cause of security vulnerabilities. For this reason, static analyzers were recently proposed for detecting crypto API misuses. They differ in strengths and weaknesses, and they might miss bugs. Motivated by the inherent limitations of static analyzers, this article reports on a study of runtime verification (RV) as a dynamic-analysis-based alternative for crypto API misuse detection. RV monitors program runs against formal specifications; it was shown to be effective and efficient for amplifying the bug-finding ability of software tests. We focus on the popular JCA crypto API and write 22 RV specifications based on expert-validated rules in a static analyzer. We monitor these specifications while running tests in five benchmarks. Lastly, we compare the accuracy of our RV-based approach, RVSec, with those of three state-of-the-art crypto API misuses detectors: CogniCrypt, CryptoGuard, and CryLogger. Results show that RVSec has higher accuracy in four benchmarks and is on par with CryptoGuard in the fifth. Overall, RVSec achieves an average F_1 measure of 95%, compared with 83%, 78%, and 86% for CogniCrypt, CryptoGuard, and CryLogger, respectively. We highlight the strengths and limitations of these tools and show that RV is effective for detecting crypto API misuses. We also discuss how static and dynamic analysis can complement each other for detecting crypto API misuses.**

***Index Terms*—Security vulnerability, crypto API misuse, runtime verification**

I. Introduction

DEVELOPERS often use cryptographic (crypto) APIs to protect sensitive data [1], but incorrect usage of crypto APIs can make software vulnerable to attack. For example, using an insecure block cipher crypto schema (e.g., AES algorithm with the ECB mode of operation)[1] might compromise system security [2], [3]. This article is motivated by the observation that developers often struggle to comprehend the low-level requirements that are needed to correctly use crypto APIs [1], [4], [5]. Also, even though recently proposed static analyzers are quite effective in detecting crypto API misuses [6], [7], [8], we show in this article that they are still limited for detecting certain types of misuses.

The threat posed by security vulnerabilities in today's world is serious enough to warrant research on complementary approaches for crypto API misuse detection. So, in this article, we study the use of runtime verification (RV) as a dynamic-analysis-based approach for detecting crypto API misuses and compare its accuracy with those of state-of-the-art static and dynamic analyzers. RV inputs are formal specifications, the code to be checked, and input data on which to run the code. An RV tool instruments the specifications into the code so that related program events are signaled at runtime and checked against the specifications. RV then outputs *violations* if the program violates any specifications.

As with any dynamic analysis, RV offers two main advantages over static analysis tools for crypto API misuse detection. First, it may be feasible to use RV to find crypto APIs misuses during testing. RV is effective and efficient for amplifying the bug-finding capability of existing software tests [9], [10], [11], [12], [13]. Second, RV can complement static analysis—which has good coverage but often *over-approximates the program behavior*, leading to false alarms [14]. Instead, a bug-free RV tool with perfect specifications generates no false positives but may have poor coverage.

The benefits of using RV for crypto API misuse detection must be balanced with the costs of writing formal specifications and of runtime overheads. We amortize formal specification costs by writing specifications for widely-used crypto APIs. So, a crypto API specification can be checked without modification on *all* clients of that API. We also measure the runtime overheads of using RV to check one version of programs, but, in practice, techniques exist that can be used to significantly speed up RV during software evolution [9], [10].

Manuscript received 30 November 2022; revised 14 July 2023; accepted 19 July 2023. Date of publication 21 August 2023; date of current version 17 October 2023. Recommended for acceptance by L. Mariani. *(Corresponding author: Rodrigo Bonifácio.)*

Adriano Torres, Pedro Costa, Luis Amaral, Jonata Pastro, Rodrigo Bonifácio, and Edna Dias Canedo are with the Computer Science Department, University of Brasília, Brasília 70910, Brazil (e-mail: adriano.torres@aluno.unb.br; teixeira.pedro@aluno.unb.br; luis.amaral@aluno.unb.br; jonata.pastro@aluno.unb.br; rbonifacio@unb.br; ednacanedo@unb.br).

Marcelo d'Amorim is with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695 USA, and also with the Informatics Center, Federal University of Pernambuco, Recife 50670, Brazil (e-mail: mdamori@ncsu.edu).

Owolabi Legunsen is with the Department of Computer Science, Cornell University, Ithaca, NY 14850 USA (e-mail: legunsen@cornell.edu).

Eric Bodden is with the Paderborn University and the Fraunhofer Institute for Mechatronic Systems Design, 33098 Paderborn, Germany (e-mail: eric.bodden@uni-paderborn.de).

Digital Object Identifier 10.1109/TSE.2023.3301660

[1]A block cipher requires a mode of operation to encrypt plain text of arbitrary length.

Evaluating the accuracy of RV is particularly challenging because manually writing specifications is notoriously difficult [15], [16], [17]. In particular, translating crypto recommendations, informally available in crypto standards, CVEs, and CWEs, for a specific crypto API and RV implementation requires a thorough validation process. Moreover, although specification miners exist [18], they are often imprecise [19] and can infer API misuses as specifications [20]. Unfortunately, our research also reveals that existing datasets of crypto API *uses* and *misuses* in the literature [21] contain test cases that fail to execute—and thus, without fixes, are inadequate to conduct experiments using dynamic analysis—and mislabel many pieces of secure code as vulnerable. These problems with existing datasets might have led to inflated performance numbers of some static analysis tools in prior work.

To obtain RV specifications, we use 22 CrySL rules [6], [20] that were previously validated by independent security experts. We manually translate these CrySL rules into 22 JavaMOP specifications. JavaMOP is a natural choice for our investigation, particularly because it is flexible to model the specifications supported by existing crypto static analyzers. For example, all 22 specifications that we write define a combination of constraints on *object state, parameters, and ordering on method calls*.

We were not able to check ordering constraints using the CryLogger tool [22]. CryLogger is a recently-proposed dynamic analysis for detecting crypto API misuses. However, CryLogger's design relies on API code instrumentation, instead of client code instrumentation, leading to crypto API warnings that one could hardly integrate into empirical assessments. Furthermore, unlike our JavaMOP approach, CryLogger lacks the capability to verify ordering constraints on method calls. This limitation hinders the CryLogger's ability to detect specific crypto API misuses, such as Missing Cryptographic Step (CWE 325) [23].

To obtain ground truth for comparison, we adapt five publicly available Java benchmarks [21], [24], [25], [26] from the literature and security organizations such as the US National Security Agency (NSA) and the Open Web Application Security Project (OWASP). These benchmarks were originally curated to compare static crypto API misuse detectors. Altogether, these benchmarks contain 801 test cases, consist of more than 350K lines of code (LOC), and are a mix of small Java programs (14–52 LOC) and open-source programs (ranging from 6,788 to 164,335 LOC). Some of these programs involve tricky and complex uses and misuses of crypto APIs, and we manually inspect and (re)label all the crypto API uses and misuses in the benchmarks.

We compare the Precision, Recall, and F_1 score of our RV-based approach (RVSec) with those of state-of-the-art static analyzers—CogniCrypt and CryptoGuard—on these benchmarks. We also evaluate the runtime overhead of RVSec and the impact of code coverage on its accuracy. Overall, RVSec achieves an average F_1 measure of 95%, compared with 83%, 78%, and 86% for CogniCrypt, CryptoGuard, and CryLogger, respectively (see Section IV, Table III). On a larger benchmark, RVSec overhead while monitoring test case executions varies between 8.64% and 56.86% (see Section VI-A, Table VII). Note that we did not apply recent optimizations that speed up RV by $5x$ during software evolution [9], [10].

```
1  public String apply(String pwd, Key key) throws Exception {
2    byte[] input = Base64.getDecoder().decode(pwd);
3    Cipher cipher = Cipher.getInstance("DESede");
4    cipher.init(Cipher.DECRYPT_MODE, key);
5    byte[] output = cipher.doFinal(input);
6    return new String(output, UTF_8);
7  }
```

Fig. 1. Example trivial crypto API misuse from Apache Meecrowave [21].

Our findings show that RV and static analyzers are complementary. That is, we found crypto misuses that only RVSec detected. The converse is also true: some misuses were detected by the static analyzers and not by RVSec. We also compare RVSec with CryLogger. Since CryLogger instruments the code of the APIs, making it hard to integrate CryLogger results into our experiments, we extend the CryLogger implementation to log program stack traces whenever calls to methods of a crypto library occur. Also, our CryLogger comparison considers quantitative and qualitative aspects of both dynamic approaches.

We make the following contributions:

(a) An in-depth study that compares RVSec with state-of-the-art tools for detecting crypto API misuses (CogniCrypt, CryptoGuard, and CryLogger). We enumerate strengths and limitations of these tools concerning their crypto API misuse detection capabilities.
(b) We highlight that static and dynamic analyses complement each other for crypto API misuse identification. In particular, we discuss situations where RVSec identified misuses that static analyzers miss (and vice-versa).
(c) A set of JavaMOP specifications for checking correct usage of the JCA crypto API in Java and Android programs. Our JavaMOP prototypes and dataset are available online: https://github.com/PAMunb/rvsec
(d) Findings that led to fixes in CogniCrypt and revisions to ground truth datasets that are used in the literature on crypto API misuse detection. We also make available an experimental package[2] for comparing static and dynamic tools for crypto API misuse detection.

II. Example

This section illustrates the problem of crypto API misuse and discusses how static and dynamic analyses detect them.

An example of crypto API misuse. Fig. 1 shows a crypto API misuse from the Apache project Meecrowave [21]. The code snippet encrypts data using the `Cipher` class from the Java Cryptography Architecture API (JCA) [27]. The code initializes a `Cipher` with the `DESede` algorithm, an implementation of the Triple DES Encryption algorithm [28] that should no longer be used in production [2], [29].

In summary, to correctly use a `Cipher`, developers should [27], [30], [31] (1) use a secure cryptographic configuration to obtain a `Cipher` instance (Line 3); (2) initialize the `cipher`

[2]https://github.com/PAMunb/RVSec-replication-package.git

object using a `key` that is consistent with the `Cipher` algorithm being used (Line 4); and (3) use a certain order of method calls (lines 3–5). Departure from these conditions on `Cipher` can result in security vulnerabilities.

Static detection of crypto API misuses. Simple static analyses (or even `grep`) can detect the crypto API misuse on line 3 in Fig. 1 because it passes a hard-coded string as the parameter to `getInstance`. In practice, however, checking crypto API usages can be non-trivial. For example, one must check whether or not the order of method calls is valid and that certain values do not propagate to certain locations.

For these reasons, recent static analyzers of Crypto API misuses, such as CogniCrypt [20] and CryptoGuard [7], were proposed. Unfortunately, even advanced static analysis tools can fail to predict the program behavior adequately—producing either false positives or false negatives. For instance, considering the code snippet in Fig. 2, CryptoGuard raises a "Found broken hash function" warning in the method `withMD5()`. However, this is a false positive warning since the main program does not call the `withMD5()` method that configures the `MDHelper` instance to use the non-recommended `MD5` hash algorithm.

CogniCrypt does not raise any alarm, though, even if we instead use the `MDHelper` class with a hypothetical call to `MDHelper.instance().withMD5().digest(str)` (not shown in the code snippet). In this case, CogniCrypt would lead to a false negative.

Dynamic detection of crypto API usage. This article investigates runtime verification (RV) as a dynamic-analysis-based alternative for detecting crypto API misuses and compares RVSec with state-of-the-art tools that detect Crypto API misuses: CogniCrypt, CryptoGuard, and CryLogger. An RV tool uses formal specifications to instrument the code. Then, at runtime, it synthesizes *monitors* that check sequences of runtime *events*, such as method calls or field accesses, against the specification. Monitors are typically automata and the event sequences are often called *traces*. We use JavaMOP for Java [15], [32], [33] as the RV tool in this article. JavaMOP was used to find hundreds of additional bugs during unit testing of many open-source projects [11], [12], [13].

Fig. 3 shows an RV specification in the JavaMOP syntax; we wrote it based on the CrySL specification for the JCA `Cipher` class. (CrySL is the CogniCrypt specification language [6], [20]). Fig. 3 shows the three main parts of JavaMOP specifications: *event definitions* (lines 6 to 13), *property* (line 15), and *handlers* (line 17). An event definition specifies what event should be signaled at runtime and where the event should be instrumented. For example, lines 6 to 7 specify that calls to `Cipher.getInstance(String)` whose arguments are valid encryption/decryption algorithms (`condition` on line 7) should be signaled. The instrumentation point should be *after* (line 6) any call to the `Cipher.getInstance(String)` method.

Properties are logical formulas over events; they describe when a trace violates or validates the specification. The Extended Regular Expression (ERE) formula on line 15, Fig. 3, defines the property; it matches traces where events `g1` or `g2` occur exactly once, followed by one or more `init` events,

```
class MDHelper {

 String algorithm;

 MDHelper withSHA384(){
   this.algorithm="SHA-384";
   return this;
 }

 MDHelper withMD5(){
   this.algorithm="MD5";
   return this;
 }

 byte[] digest(String input) throws Exception {
  MessageDigest md = MessageDigest.getInstance(algorithm);
  md.update(input.getBytes());
  return md.digest();
 }

 static MDHelper instance() {...}
 ...
}

class Main {
  /* ... */
  public static void main(String args[]) {
    byte hash = MDHelper.instance().withSHA384().digest(str);
    /* ... */
  }
}
```

Fig. 2. Example `MessageDigest` usage. It is only safe to call `digest()` after configuring the helper class by calling `withSHA384()`.

and exactly one `doFinal` event. We mainly use EREs while writing the RVSec specifications because it is the property specification language that CrySL uses. Still, JavaMOP can monitor properties written in other formalisms, such as Context Free Grammar (CFG), Finite State Machine (FSM), or Linear Temporal Logic (LTL). In a few cases where the specification involved many events, we considered the FSM formalism a better alternative, leading to an easier-to-understand specification in those cases.

Handlers allow specifying arbitrary Java code that executes when a trace violates or matches the property. For example, the `@fail` handler on line 17 indicates that when a trace does not match the ERE, an error should be reported and the monitor should be reset. JavaMOP specifications are *parametric* [34], [35]. So, one monitor will be synthesized for every unique set of specification parameters. Since `Cipher` is the only specification parameter (line 1, Fig. 3), JavaMOP synthesizes one monitor per instance of the `Cipher` class during execution. Although RVSec and the two static analyzers correctly detect the crypto API misuse in the code of Fig. 1, RVSec has better precision and recall when analyzing the code in Fig. 2.

As we mentioned, in this article we also compare RVSec with CryLogger, a dynamic analysis approach for detecting crypto API misuses. Overall, CryLogger involves two main components: a new implementation of the JCA classes that logs usage information of the API and a Python module that processes the log files and outputs crypto API misuses. We changed both components to allow a fair comparison with CryLogger in our research. First, we changed the implementation of the JCA

```
public CipherSpec(Cipher c) {
  private Cipher ciph;
  private ExecutionContext ctx = ExecutionContext.instance();
  private ErrorHandler eh = ErrorHandler.instance();
  // event definitions
  event g1 after(String algMode) returning(Cipher c):
    call(public static Cipher Cipher.getInstance(String)) && args(algMode) && condition(isValid(algMode)) { ciph = c; }
  event g2 after(String algMode, String p) returning(Cipher c):
    call(public static Cipher Cipher.getInstance(String, String)) && args(algMode, p) && condition(isValid(algMode)) { ciph = c; }
  event init before(int mode, Key key, Cipher c): call(public void Cipher.init(int, Key,..)) && args(mode, key, ..) &&
    target(c) && condition(ctx.validate(Property.GENERATED_KEY, key)){ }
  event doFinal after(Cipher c) returning(byte[] ciphText): call(public byte[] Cipher.doFinal(..)) && target(c)
    { ctx.setProperty(Property.ENCRYPTED, ciphText); }
  // properties
  ere: (g1 | g2) init+ doFinal
  // handlers
  @fail { eh.reportError(); __RESET; }  }
```

Fig. 3. Example of a JavaMOP specification for the JCA `Cipher` class.

TABLE I
BENCHMARKS IN OUR STUDY

Benchmark	Used In	TCs	SLOC	Misuses
MASCBench	[24]	30	0.5K	28
SmallCryptoAPIBench	[21], [38], [39]	187	3.7K	130
OWASPBench	[40], [41], [42]	482	47.5K	259
JulietBench	[43], [44]	102	6.8K	102
ApacheCryptoAPIBench	[21], [38], [39]	-	303.3K	74

TABLE II
DETAILS OF THE APACHECRYPTOAPIBENCH, HIGHLIGHTED IN TABLE I

Project	Module	Revision	TCs	SLOC	Misuses
Artemis	Artemis-commons	5ab187b	110	11,737	15
Dir. Server	Apacheds-kerberos-codec	155bd94	376	42,185	19
ManifoldCF	Mcf-core	9573dc4	5	21,281	3
DeltaSpike	Deltaspike-core-impl	d95abe8	155	13,515	2
Meecrowave	Meecrowave-core	3780f1c	19	6,788	3
Spark	Spark-core_2.11	9ff1d96	2,045	164,335	27
Tika	Tika-core	6f33bae	222	23,207	0
Wicket	Wicket-util	dbd86d9	237	20,220	5

classes to log not only usage information of the crypto API, but also the JVM stack trace. Second, we changed the Python module to output not only the existence of a crypto API misuse, but also the client methods that might originate a call to a crypto API. Without these changes, we would not be able to integrate the outputs of CryLogger into our study settings.

III. STUDY SETTINGS

We study the strengths and weaknesses of RVSec for detecting crypto API misuses. In particular, we quantitatively compare the accuracy of RVSec and state-of-the-art tools based on static and dynamic analyses (CogniCrypt, CryptoGuard, and CryLogger), and qualitatively assess the main reasons for inaccuracy. Our goal is to investigate whether RVSec is more beneficial than existing tools for detecting misuses of crypto APIs and to characterize reasons why RVSec and the other tools generate false positives or miss to detect crypto API misuses.

A. Benchmarks

Table I shows the benchmarks that we used in our evaluation. **MASCBench** contains 30 small Java programs with crypto API misuses [24]. The programs are from open-source Android apps and Apache Qpid Brokerj [36]; they are minimized to only show crypto API misuses. **SmallCryptoAPIBench** contains 187 test cases for legal and illegal usages of crypto APIs. We obtain SmallCryptoAPIBench by removing 16 non-JCA test cases from the Afrose et al. [21] benchmark. **OWASPBench** contains 482 test cases related to crypto APIs. The test suite is curated by OWASP (https://owasp.org) [25]. We use the tests that are related to JCA and cover the common crypto API misuses CWE-327 (Use of a Broken or Risky Cryptographic Algorithm) and CWE-328 (Reversible One-Way Hash). **JulietBench** contains 102 test cases curated by the US National Security Agency (NSA) [26], [37]. The breakdown of tests per CWE is: 17 CWE-325 (Missing Cryptographic Step), 34 CWE-327, and 51 CWE-328. **ApacheCryptoAPIBench** contains real-world crypto API misuses from Apache projects [21]. The original benchmark involves eight Apache projects with different degrees of test coverage. Those projects include a total of 3169 test cases, but not all of them are JCA-related. We present more details of the ApacheCryptoAPIBench in Table II. The CryLogger executions do not finish for three ApacheCryptoAPIBench projects: Artemis, Spark, and Tika.

B. Procedure and Metrics

To evaluate the accuracy of the tools, we compute Precision, Recall, and F_1 score—metrics that are typically used in related work [21], [39]. We compute them as follows:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3}$$

Besides accuracy, we conduct further analysis using ApacheCryptoAPIBench only. ApacheCryptoAPIBench comprises open-source systems, which allows us to better understand the RVSec implications on overhead in more realistic scenarios. For instance, to estimate the overhead of RVSec while executing the test suites, we run the

ApacheCryptoAPIBench tests ten times with and without RV, and average the execution times. TRV and $TBase$ are the average time to run the test cases with RV and without it, respectively. The overhead of RV corresponds to the percentage increase of time of TRV over $TBase$ (=$100 \times (TRV - TBase)/TBase$).

We also measure statement, branch, and method coverage in ApacheCryptoAPIBench using the JaCoCo tool [45]. Our goal is to investigate how code coverage impacts the accuracy and overhead of RVSec for detecting crypto API misuses.

C. RV of the JCA Crypto API Using JavaMOP

We write JavaMOP JCA specifications to check the correct usage of the API. Fig. 3 is an example of JavaMOP specification that we write. To write JavaMOP specifications, we use an existing set of CrySL JCA specifications as our starting point for three main reasons. First, the CrySL JCA specifications have been validated by crypto experts. Second, there are similarities between the CrySL and JavaMOP specification languages. Third, a test suite provided by the CogniCrypt development team (with 31 JUnit test classes and over 200 test methods) allowed us to more easily validate the JavaMOP specifications that we write.

We create a custom infrastructure for performing RV during unit testing, so that we could reuse the CogniCrypt test suite almost as-is. The few fixes that we make on the CogniCrypt test suite relate to incorrect configuration of keys, cipher algorithms, and operation modes (e.g., decrypt or encrypt) that caused runtime exceptions.

The CrySL repository [46] provides 47 rules for JCA [20]. Still, previous studies show that a subset of 12 JCA classes (including `MessageDigest`, `Cipher`, `Signature`, and `KeyGenerator`) is more frequently used and leads to most crypto API misuses [20], [47]. So, we start writing JavaMOP specifications for these 12 JCA classes and ended up with ten additional JavaMOP specifications the core classes depend on. In total, we wrote 22 JavaMOP specifications used in our study.

IV. Accuracy of Techniques

This section presents quantitative results to compare the accuracy between RVSec and the other crypto API misuse detectors, CogniCrypt, CryptoGuard, and CryLogger. Table III reports the performance of each technique on the five benchmarks.

A. MASCBench: For MASCBench, all four tools have 100% Precision; the "0s" in the "FP" columns are an indication that no technique reports a false positive. RVSec also did not miss any crypto API misuse (leading to a Recall of 100%), while CogniCrypt, CryptoGuard, and CryLogger missed five, nine, and eight misuses, respectively. The false negatives in CogniCrypt and CryptoGuard are due to different limitations in the static analyses they implement. For instance, if a call to `keyGenerator.getInstance()` returns the unsafe crypto schema, "AES" (mapped to the schema "AES/ECB/PKCS5Padding"), both static analysis tools fail to identify the crypto misuse in statements with the code `Cipher.getInstance(keyGenerator.getInstance());`. False negatives in CryLogger occur because the tool fails to detect when an Initialization Vector Parameter Spec is initialized using a constant array of bytes (accounting for four CryLogger false negatives). See an example in Fig. 4. Three other CryLogger false negatives are due to test cases that incorrectly seed secure random (see an example in Fig. 5). We provide more details in Section V.

TABLE III
Accuracy Results. Note: This Table Does Not Present the CryLogger Accuracy Results for the ApacheCryptoAPIBench—Essentially Because the CryLogger Experiment Finished Only for Five (Out of Eight) ApacheCryptoAPIBench Projects

	TP	FP	FN	Precision	Recall	F_1
			MASCBench			
RVSec	28	0	0	1.00	1.00	1.00
CogniCrypt	23	0	5	1.00	0.82	0.90
CryptoGuard	19	0	9	1.00	0.67	0.80
CryLogger	20	0	8	1.00	0.71	0.83
			SmallCryptoAPIBench			
RVSec	122	6	8	0.95	0.93	0.94
CogniCrypt	106	24	24	0.81	0.81	0.81
CryptoGuard	114	18	17	0.86	0.87	0.86
CryLogger	98	13	32	0.88	0.75	0.81
			OWASPBench			
RVSec	259	1	0	0.99	1.00	0.99
CogniCrypt	259	201	0	0.56	1.00	0.72
CryptoGuard	219	27	40	0.89	0.84	0.86
CryLogger	317	53	0	0.82	1.00	0.92
			JulietBench			
RVSec	100	0	2	1.00	0.98	0.99
CogniCrypt	102	60	0	0.62	1.00	0.77
CryptoGuard	85	0	17	1.00	0.83	0.90
CryLogger	86	7	16	0.92	0.84	0.88
			ApacheCryptoAPIBench			
RVSec	39	1	12	0.97	0.76	0.85
CogniCrypt	48	10	3	0.82	0.94	0.88
CryptoGuard	20	14	31	0.58	0.39	0.47
Average			RVSec	0.98	0.93	0.95
			CogniCrypt	0.76	0.91	0.83
			CryptoGuard	0.87	0.72	0.78
			CryLogger	0.90	0.82	0.86

```
1 public class BaseStaticIV {
2   public static void main(String[] args) {
3     byte[] bytes = "Hello".getBytes();
4     IvParameterSpec ivSpec = new IvParameterSpec(bytes);
5     System.out.println(new String(ivSpec.getIV()));
6     System.out.println(new String(bytes));
7   }
8 }
```

Fig. 4. Example misuse due to wrongly instantiating the class `IvParameterSpec` using a constant byte array.

B. SmallCryptoAPIBench: Although RVSec does not achieve optimal performance (i.e., F_1=1.0) in this benchmark, its accuracy is still superior compared to the other tools. Seven (of eight) misuses that RVSec misses relate to the use of hard-coded passwords for loading key stores (e.g., lines 3 and 6 in Fig. 6).

Failure to handle hard-coded strings is a limitation of RVSec—we cannot check at runtime whether a variable has been initialized to a hard-coded string constant. The CryLogger false positives in the SmallCryptoAPIBench are due to overly strong constraints. For instance, CryLogger signals

```
1  public class SecureRand03 {
2    public static void main(String[] args) throws Exception {
3      byte[] seedBytes = "Seed".getBytes(StandardCharsets.UTF_8);
4
5      //"The SecureRandom instance is seeded with the
6      // specified seed bytes." --> unsafe
7
8      SecureRandom rand1 = new SecureRandom(seedBytes);
9      SecureRandom rand2 = new SecureRandom(seedBytes);
10
11     System.out.println(rand1.nextInt()==rand2.nextInt());
12   }
13 }
```

Fig. 5. Example misuse due to incorrectly seeding secure randoms.

```
1  public class PredictableKeyStorePasswordABICase1 {
2    public static void main(String args[]) throws Exception {
3      String key = "password";
4      KeyStore ks = KeyStore.getInstance("JKS");
5      URL cacerts = new File("testInput-ks").toURI().toURL();
6      ks.load(cacerts.openStream(), key.toCharArray());
7    }
8  }
```

Fig. 6. Example misuse due to a hard-coded password that RVSec misses, but CogniCrypt and CryptoGuard detect.

```
1  String alg = "";
2  java.util.Properties ps = new java.util.Properties();
3  ps.load(...getResourceAsStream("benchmark.properties"));
4  alg = ps.getProperty("cryptoAlg2", "AES/ECB/PKCS5Padding");
5  javax.crypto.Cipher c = javax.crypto.Cipher.getInstance(alg);
```

Fig. 7. OWASPBench code where CryptoGuard reports a false positive.

warnings even for safe `Cipher` crypto-schemes (a string in the format ALGORITHM/MODE/PADDING such as `AES/CBC/PKCS5Padding`.). Further, CryLogger was the tool that generated the highest number of false negatives for the SmallCryptoAPIBench. The main reason is that CryLogger does not identify the use of predictable seeds and credentials in strings. We further exemplify and analyze the false positives and false negatives from all tools in Section V.

C. OWASPBench: Recall from Table I that OWASPBench is the benchmark with the highest number of test cases manifesting API misuses. RVSec again achieves the highest accuracy (F_1 score = 0.99) in this benchmark, compared to the other tools. RVSec, CogniCrypt, and CryLogger did not miss any crypto API misuse. However, CogniCrypt reports 201 false positives on this benchmark, severely affecting its Precision (56%). We observe that test classes in OWASPBench often contain a control flow path that does not execute all sequences of method calls that CogniCrypt expects (according to the CrySL rules); CogniCrypt reports a false positive warning for all these cases. Differently, CryptoGuard reports 27 false positives; they are all related to test cases that load a crypto schema using a configuration file. Fig. 7 shows an illustrative code snippet.

At runtime, the call to the `getProperty` method in Fig. 7 retrieves the valid crypto schema `AES/GCM/NoPadding`, even though CryptoGuard wrongly approximates that an alternative value, `AES/ECB/PKCS5Padding`, is being assigned. Similarly, CryptoGuard assumes valid algorithms that appear as alternatives to invalid ones in a

```
cryptoAlg1=DES/ECB/PKCS5Padding
cryptoAlg2=AES/GCM/NoPadding
hashAlg1=MD5
hashAlg2=SHA-256
testCases.per.folder=80
testsuite-version=1.2
```

Fig. 8. A configuration file that OWASPBench uses at runtime.

configuration file, leading CryptoGuard to miss 40 crypto misuses. For instance, CryptoGuard detects from the statement `ps.getProperty("hashAlg1", "SHA512")` that the secure algorithm `SHA512` is in use. However, OWASPBench uses a configuration file assigning an unsafe hashing algorithm, `MD5`, to the string `"hashAlg1"` as shown in Fig. 8.

We observe that the use of configuration files is a major source of CryptoGuard imprecision, leading to many false positives and false negatives in OWASPBench. Instead, the CryLogger assessment did not reveal any false negatives (recall of 100%), though it generated 53 false positives. Our manual analysis revealed that these false positives are also motivated by overly strong constraints on the CryLogger rules that raise warnings for safe crypto schemes.

D. JulietBench: RVSec also achieves higher F_1 score than static techniques on JulietBench. For Recall, RVSec misses two misuses while CryptoGuard misses 17 misuses, CryLogger misses 16, and CogniCrypt does not miss any. After a manual analysis, we find that the two crypto API misuses that RVSec misses are due to non-determinism in the choice of the hashing algorithm used; one is secure and the other is not. Our experiments execute the secure choice and missed the misuse. Research on dealing with test non-determinism (or flakiness) is receiving a lot of attention [48], [49], [50], [51]; these results could be used in the future to mitigate the impact of non-deterministic runs during RVSec. CryLogger suffers from the same limitation, and five CryLogger false positives are due to non-deterministic choices that lead the program examples to execute an insecure path labeled as secure in the benchmark ground truth.

The 17 misuses that CryptoGuard misses relate to CWE-325 (Missing Required Cryptographic Step). CryptoGuard has no support for detecting misuses that are due to an invalid or incomplete method-call sequence. Similarly, all 16 CryLogger false negatives are due to incomplete method-call sequences. Finally, RVSec and CryptoGuard do not report any false positive, even though CogniCrypt reports 60; all of them related to CWE-327 (Use Broken Crypto). The reason for false positives is that CogniCrypt does not consider secure the code idiom below:

```
SecretKey secretKey = keyGenerator.generateKey();
byte[] key = secretKey.getEncoded();
SecretKeySpec secretKeySpec = new SecretKeySpec(key, "AES");
```

that instantiates a key byte array using the `getEncoded()` method in the `SecretKey` class. All false positives that CogniCrypt reports for JulietBench are due to this idiom.

E. ApacheCryptoAPIBench: Unfortunately, the provided ground truth in ApacheCryptoAPIBench misses essential information that we need to compute Precision and Recall.

TABLE IV
SUMMARY OF WARNINGS TECHNIQUES REPORT ON APACHECRYPTOAPIBENCH

Tool	Full Data Set	Curated Data Set
RVSec	64	40
CogniCrypt	72	58
CryptoGuard	56	36

For example, many *true positives* in the original ApacheCryptoAPIBench ground truth do not specify the Java class in which a crypto API misuse occurs. We also found and shared with the authors of the ApacheCryptoAPIBench possible situations where the labels assigned to the warnings in the ground truth were incorrect. Posteriorly, they agreed with almost all observations we shared and changed the benchmark.

Despite these threats in the original ApacheCryptoAPIBench ground truth, the ApacheCryptoAPIBench benchmark can still be a valuable source of data about how RVSec compares with the other tools on real-world open-source projects—where the static detectors were previously evaluated. For this reason, we manually inspect all warnings generated by RVSec, CogniCrypt, and CryptoGuard on ApacheCryptoAPIBench and generate our own ground truth. We count as unique all warnings from a given class/method. This decision is necessary because the techniques may generate different numbers of warnings for a given misuse.

In the end, we obtain 192 unique warnings. Of these, we remove (a) 44 warnings that originate from third-party libraries, and (b) 14 warnings that RVSec reports in JUnit test code (to make a fair comparison with CogniCrypt and CryptoGuard, which do not analyze test classes). Our final dataset contains 134 warnings, summarized in Table IV. Creating our own ground truth may introduce threats to validity, but doing so allows us to avoid inconsistencies that we found in the provided ApacheCryptoAPIBench ground truth. The reason that led us not to consider the CryLogger outputs to build our ApacheCryptoAPIBench ground truth is twofold. First, the CryLogger execution finished for only five out of eight projects in the ApacheCryptoAPIBench. Second, CryLogger generated thousands of warnings for the ApacheCryptoAPIBench, many reporting crypto API misuses in the standard Java classes—and not in the application classes. It is unfeasible to manually validate these warnings without implementing additional features to CryLogger.

Table III shows the Precision, Recall, and F_1 score using the ground truth dataset that we curate for ApacheCryptoAPIBench. Although RVSec has the highest Precision, when we consider F_1 score, CogniCrypt performs better. Since the CryLogger experiment does not finish the execution for three ApacheCryptoAPIBench projects, CryLogger leads to the highest number of false negatives (34) in our investigation. This number of false negatives would be lower if CryLogger had finished the execution for all projects. Conversely, the number of false positives CryLogger reports is somewhat high: eight false positives in total. This number would likely increase if CryLogger had finished the execution for all projects. So, we do not summarize these results in Table III.

```
1  void parse( InputStream stream, ContentHandler handler,
2              Metadata metadata, ParseContext context) ... {
3    cipher = Cipher.getInstance(transformation);
4    Key key = context.get(Key.class);
5    AlgoParams p = context.get(AlgoParams.class);
6    SecureRandom random = context.get(SecureRandom.class);
7
8    if (p != null && random != null) {
9      cipher.init(Cipher.DECRYPT_MODE, key, p, random);
10   } else if (p != null) {
11     cipher.init(Cipher.DECRYPT_MODE, key, p);
12   } else { cipher.init(Cipher.DECRYPT_MODE, key); }
13   super.parse(
14     new CipherInputStream(stream, cipher),
15     handler, metadata, context);
16 }
```

Fig. 9. Code snippet from the Tika project. In this case, a `Cipher` is being just prepared to future usage.

Regarding CryptoGuard false positives, the tool implements fewer rules than CogniCrypt, which is the main reason that leads CryptoGuard to find only 58% of the ApacheCryptoAPIBench misuses. RVSec misses 12 of 48 misuses that CogniCrypt detects. We find that these false negatives are due to (a) lack of test cases necessary to reveal the issues, and (b) lack of RV specifications for less frequently used JCA classes (e.g., `SecretKeyFactory` and `TrustManagerFactory`). We will specify additional rules for RVSec in future work.

With respect to false positives, RVSec reports one, while CogniCrypt, CryptoGuard, and CryLogger reports 10, 14 and 13, respectively. Thirteen false positives from CryptoGuard result from a rule that approximates the `java.util.Random` class to be insecure. Nonetheless, `java.util.Random` is often used in *non*-cryptographic contexts. After careful manual analysis, we find no instance of the `java.util.Random` class in the ApacheCryptoAPIBench that leads to a vulnerability. After confirming this conclusion with the authors of the ApacheCryptoAPIBench [21], we mark these examples as false positives. If we consider these usages of the `java.util.Random` true positives, CryptoGuard Precision on ApacheCryptoAPIBench would be higher at 0.95. We keep these warnings in our dataset because previous research wrongly labeled them as true positives, causing misleading results to be published [21], [39]. Similarly, the safe usage scenarios of `java.util.Random` are the main source of false positives for CryLogger in the ApacheCryptoAPIBench.

Our manual inspection reveals that the 10 false positives that CogniCrypt reports involve tricky situations. For example, in Tika, code that prepares a `Cipher` may not explicitly call methods such as `update` or `doFinal` as shown in Fig. 9. Currently, CogniCrypt reports an error, but we find that developers intend for such missing calls to be made by clients of the code—what happens at runtime. We label these as false positives. We summarize our findings on accuracy below.

Summary: Results shows that RVSec is very accurate ($\bar{F}_1$=0.95), compared to CryLogger ($\bar{F}_1$=0.85), CogniCrypt ($\bar{F}_1$=0.83), and CryptoGuard ($\bar{F}_1$=0.78).

TABLE V
ACCURACY (F_1 SCORE) OF THE TOOLS WITH RESPECT TO CWEs (COMMON WEAKNESS ENUMERATION)

CWE	RVSec	CogniCrypt	CryptoGuard	CryLogger
321 - Use of Hard-coded Cryptographic Key	0.86	0.94	0.22	0.00
325 - Missing Cryptographic Step	1.00	0.15	0.00	0.10
327 - Use of a Broken or Risky Cryptographic Algorithm	0.99	0.90	0.91	0.90
328 - Use of Weak Hash	0.99	0.97	0.88	0.94
337 - Predictable Seed in Pseudo-Random Number Generator	1.00	0.00	0.85	0.00
338 - Use of Cryptographically Weak Pseudo-Random Number Generator	1.00	1.00	0.47	0.67
341 - Predictable from Observable State	0.93	0.94	0.89	0.00
798 - Use of Hard-coded Credentials	0.84	0.84	0.88	0.90
916 - Use of Password Hash With Insufficient Computational Effort	0.84	0.84	0.67	0.93
1204 - Generation of Weak Initialization Vector (IV)	0.97	1.00	0.90	0.65
1240 - Use of a Cryptographic Primitive with a Risky Implementation	0.75	0.89	0.00	0.00
1391 - Use of Weak Credentials	0.77	0.86	0.75	0.77

V. WHAT ARE THE CAUSES OF INACCURACY?

This section discusses the causes of inaccuracy (i.e., false positives and false negatives) for RVSec (§V-A), for the static analyzers CogniCrypt and CryptoGuard (§V-B), and for CryLogger §V-C. Table V summarizes the accuracy of the tools with respect to CWEs.

A. Sources of Inaccuracy: RVSec

As expected for a dynamic analysis approach, the precision of RVSec is very high. The rare cases of false positives (8 in 556 reports; Table III) are due to an overly constrained specification that expects at least 10 thousand iterations when using Password Based Encryption—differently, the ground truth for SmallCryptoAPIBench labels at least 1000 iterations as secure (CWE-1391). Overall, of the 22 false negatives associated with RVSec (Table III, column "FN"), ten can be attributed to a lack of test inputs to exercise the crypto API misuses in ApacheCryptoAPIBench and twelve can be attributed to inherent limitations of dynamic analysis (e.g., RVSec fails to detect the use of constant strings to initialize cryptographic primitives).

1) Lack of Test Inputs: It is expected that low-quality test suites in RV can lead to false negatives, whereas over-approximation by static analysis can lead to false positives. The handcrafted benchmarks MASCBench, SmallCryptoAPIBench, OWASPBench, and JulietBench contain suites of test cases that directly execute code with and without crypto API misuses. As such, the high coverage of test suites in these benchmarks might justify the high accuracy of RVSec. Regarding these four handcrafted benchmarks, every crypto API usage appears in one test scenario—leading to a complete coverage of the code that use the JCA crypto API.

Accordingly, we further investigate the problem about lack of test inputs with a coverage assessment on the ApacheCryptoAPIBench open-source projects. To this end, we measure four test suite metrics: Number of Test Cases (TCs), average Instruction Coverage (IC), average Branch Coverage (BC), and average Method Coverage (MC)—the last three are coverage criteria that we measure using the Java Code Coverage Library (JaCoCo), which we integrated into the build process of the ApacheCryptoAPIBench projects. After running the test suites, JaCoCo exports test coverage measurements for each class of a project, from which we computed the average coverage (i.e., instruction, branch, and method level coverage) presented in Table VI.

TABLE VI
SUMMARY OF THE TEST SUITE METRICS

Apache Module	TCs	IC	BC	MC
Dir. Server	376	73.79	44.7	71.19
Artemis	110	29.41	31.24	35.46
ManifoldCF	5	8.29	6.56	9.27
Meecrowave	19	65.01	46.36	56.56
DeltaSpike	155	69.86	61.23	84.1
Spark	2,045	23.25	17.03	22.13
Tika	222	48.63	49.92	50.28
Wicket	237	39.55	37.63	40.98

We find the coverage of test suites to be moderate on the ApacheCryptoAPIBench: average of 44.72% instruction coverage (IC). Since we do not augment the test suites in ApacheCryptoAPIBench, we expected that RVSec would only find a substantially smaller number of warnings than CogniCrypt and CryptoGuard find. However, RVSec misses only 12 of 51 crypto API misuses in the ApacheCryptoAPIBench, regardless of the moderate coverage in the benchmark. The main reason for RVSec's 0.76 recall on ApacheCryptoAPIBench—despite the 44.72% instruction coverage—is that crypto API is confined to a few of the projects' classes, which are covered by the tests. We find a lower RVSec recall in projects where crypto API usage is not covered by tests. For example, *RVSec missed all 3 API misuses from project ManifoldCF* (Table II) because that project did not contain tests executing the methods with crypto API misuses.

> *Effect of Test Coverage in RVSec's performance:* RVSec achieves 0.76 recall on ApacheCryptoAPIBench although the tests cover only 44.72% of the instructions, on average. That happens because usage of the crypto API is confined to a few covered classes, which are exercised by the tests.

2) Inherent Limitation of RVSec: Seven (out of eight) RVSec's false negatives in SmallCryptoAPIBench are due to the use of hard-coded passwords for loading key stores. According to CWE-798, hard-coded passwords can be a threat since "*hard-coded credentials typically create a significant hole that allows*

```
1  public class PredictableKeyStorePassword {
2    public void go() throws Exception {
3      String type = "JKS";
4      KeyStore ks = KeyStore.getInstance(type);
5      cacerts = new File("input-ks").toURI().toURL();
6      String defaultKey = "password";
7      /* error: defaultKey is hard-coded */
8      ks.load(cacerts.openStream(), defaultKey.toCharArray());
9    }
10 }
```

Fig. 10. An example false negative from RVSec.

an attacker to bypass the authentication that has been configured by the software administrator" [52]. It is very difficult to write a specification that allows RVSec to check at runtime if the string being used as a password was hard-coded at initialization (see Lines 6 and 8 in Fig. 10). The recommended best-practice is to retrieve the passwords from an external and protected file or database [52]. This is a situation where we could benefit from complementing RVSec with static detectors like CogniCrypt and CryptoGuard.

> *Main reason for false negatives in RVSec:* It is hard to write RVSec specifications for checking if a variable was assigned a hard-coded constant string at initialization.

B. Sources of Inaccuracy: Static Analyzers

Fig. 11 shows a crypto API misuse that neither CogniCrypt nor CryptoGuard detected, but which RVSec and CryLogger detected. There, `Cipher` is instantiated by calling the `getAlgorithm` method of class `KeyGenerator`. In the example, `getAlgorithm` returns the `String "AES"`—since `keygen` is instantiated using a call to `KeyGenerator.getInstance("AES")`. But, instantiating the `Cipher c` in this way is similar to calling `Cipher.getInstance("AES")`, which specifies just the cipher algorithm, and not its operation mode and padding. The vulnerability occurs because the default mode and padding configuration for `AES` is `ECB/PKCS5Padding`, which might result in disclosing of sensitive information [29]. So, creating a `Cipher` as shown is insecure, but CogniCrypt and CryptoGuard do not detect this misuse (CWE-327). However, if one passes the `"AES"` string to the `Cipher.getInstance` method directly, instead of calling `KeyGenerator.getAlgorithm()`, both tools detect the misuse. To know that `keygen.getAlgorithm()` returns `"AES"`, both CogniCrypt and CryptoGuard should be enriched with field sensitive data flow analysis or by explicitly modeling this API call manually.

CogniCrypt and CryptoGuard also miss crypto API misuses in the scenarios that use multiple method calls to initialize a `Cipher` class. Fig. 12 presents an example, where the test case initializes a `Cipher c` instance using the insecure "DES" algorithm (CWE-327). Note that CryptoGuard does not report any issue with the test case of Fig. 12. Although CogniCrypt also misses the first error in this test case (Line 13), it correctly detects the second one (Line 19). Extending CryptoGuard

```
public class CipherExample09 {
  public static void main(String[] args) {
    try{
      KeyGenerator keygen = KeyGenerator.getInstance("AES");
      SecretKey key = keygen.generateKey();
      /* error */
      Cipher c = Cipher.getInstance(keygen.getAlgorithm());
     /* possible patch:
      * Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
      */
      c.init(Cipher.ENCRYPT_MODE, key);
      c.doFinal("something".getBytes());
    } catch(Exception e ){ e.printStackTrace(); }
  }
}
```

Fig. 11. Program in MASCBench that yields false negatives in CogniCrypt and CryptoGuard.

```
1  public class Ex05 {
2    private String cName = "AES/GCM/NoPadding";
3    privatet String name = "";
4    public Ex05 a(){ cName = "AES/GCM/NoPadding"; return this; }
5
6    public Ex05 b(){ cName = "DES"; return this; }
7
8    public String getCipherName(){ return cName; }
9
10   public static void main(String[] args) throws Exception {
11     name = new Ex05().a().b().getCipherName();
12     /* error: DES is not secure */
13     Cipher c = Cipher.getInstance(name);
14     runCipher(c);
15   }
16
17   public static void runCipher(Cipher c) throws Exception {
18     /* error: DES is not secure */
19     Key key = KeyGenerator.getInstance("DES").generateKey();
20     c.init(Cipher.ENCRYPT_MODE, key);
21     byte[] cipherText = c.doFinal("password".getBytes());
22   }
23 }
```

Fig. 12. Example of CryptoGuard false negative for MASCBench.

and CogniCrypt with a more advanced inter-procedural data flow analysis might reduce these inter-procedural false negative scenarios in both tools. For instance, FlowDroid is able to detect source-sink flows using different method calls and string manipulations [53].

Both static analyzers (CogniCrypt and CryptoGuard) report false positives for the SmallCryptoAPIBench path-sensitive programs. Fig. 13 shows an example. There, `choice` is initialized to the constant value `2`, so the condition on line 6 is always true and the secure `SHA-256` algorithm is always used on line 7, as expected. But the over-approximation that CogniCrypt and CryptoGuard employ make them flag lines 8 and 9 of Fig. 13 as places where the `md` instance of the `MessageDigest` class *may* be using an insecure implementation (CWE-328). It is questionable whether code similar to the one in Fig. 13 appears in real-world projects.

CogniCrypt reports 201 false positives in OWASPBench. We manually analyze 20 of these false positives, selected randomly. In all the cases we analyze, there is a path in the source code examples that does not satisfy the expected sequence of events that CrySL rules specify for the `Cipher` or `MessageDigest`

```
1  public class BrokenHashABPSCase1 {
2    public static void main (String [] args) throws Exception {
3      String name = "abcdef";
4      int choice = 2;
5      MessageDigest md = MessageDigest.getInstance("SHA1");
6      if(choice>1)
7        md = MessageDigest.getInstance("SHA-256");
8      md.update(name.getBytes());
9      System.out.println(md.digest());
10   }
11 }
```

Fig. 13. Path sensitive example that leads to false positives in both CogniCrypt and CryptoGuard.

```
1 MessageDigest md = MessageDigest.getInstance("sha-384");
2 if(condition()) { return; }
3 md.update(SECRET.getBytes());
4 byte[] hash = md.digest();
```

Fig. 14. OWASPBench code where CogniCrypt reports false positives.

classes (CWE-325). The code in Fig. 14 illustrates the situation, for which CogniCrypt reports a warning like:

> IncompleteOperationError: violating CrySL rule for java.security.MessageDigest. Operation on object of type `java.security.MessageDigest` object not completed. Expected call to `digest` or `update`.

Removing the `if` statement on line 2 in Fig. 14 will cause CogniCrypt to no longer report a warning. In this case, the resulting path calls the `update` and `digest` methods of the `MessageDigest` class, which matches the expected sequence of method calls in the CrySL rule for `MessageDigest`. CryptoGuard also performs poorly in detecting CWE-325 misuses (Missing Cryptographic Step). The main reason is that there is no CryLogger rule for detecting this crypto API misuse.

Considering OWASPBench, CryptoGuard reports 27 false positives and 40 false negatives—all these misclassifications are related to wrong assumptions that CryptoGuard makes when an invalid algorithm identifier can flow to the instantiation of a JCA crypto primitive (e.g., a `Cipher` or a `MessageDigest`). We also isolate these problems in small test cases (see listings in Figs. 15 and 16). So, assuming we setup a configuration file with the following content:

```
cipher01=AES/GCM/NoPadding
cipher02=AES/ECB/PKCS5Padding
```

Since CryptoGuard does not account for configuration files (this is a challenge for static analysis in general), it wrongly labels the code in Fig. 15 as a crypto API misuse (according to the OWASPBench ground truth). For the same reason, CryptoGuard wrongly classifies the code in Fig. 16 as secure—the call to `ps.getProperty()` returns `AES/ECB/PKCS5Padding`, which is not recommended (CWE-327). CogniCrypt does not assume anything when the crypto algorithm definitions come from configuration files, so it does not raise any warning about `Cipher` instantiations in Figs. 15 and 16.

```
Properties ps = new Properties();
ps.load(new FileReader(FILE_NAME));
String alg = ps.getProperty("cipher01", "AES/ECB/PKCS5Padding");
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
SecretKey key = keyGenerator.generateKey();
Cipher c = Cipher.getInstance(alg);
c.init(Cipher.ENCRYPT_MODE, key);
byte[] result = c.doFinal(SECRET.getBytes());
```

Fig. 15. Scenario for which CryptoGuard wrongly assumes that the unsafe algorithm configuration `AES/ECB/PKCS5Padding` flows to the call to the `Cipher.getInstance` method. This is an example of CryptoGuard false positive.

```
Properties ps = new Properties();
ps.load(new FileReader(FILE_NAME));
String alg = ps.getProperty("cipher01", "AES/GCM/NoPadding");
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
SecretKey key = keyGenerator.generateKey();
Cipher c = Cipher.getInstance(alg);
c.init(Cipher.ENCRYPT_MODE, key);
byte[] result = c.doFinal(SECRET.getBytes());
```

Fig. 16. Scenario for which CryptoGuard wrongly assumes that the safe algorithm configuration `AES/GCM/NoPadding` flows to the call to the `Cipher.getInstance` method. This is an example of CryptoGuard false negative.

> Scenarios involving (a) method calls with string manipulation, (b) path and field sensitivity, and (c) the use of configuration files are the main causes of inaccuracy for the static detectors.

C. Sources of Inaccuracy: CryLogger

Our assessment reports that CryLogger generates 53 false positives for the OWASPBench. After a careful analysis, we conclude that most of the CryLogger false positives are due to rules that raise warnings when using safe crypto schemes, such as "RSA/ECB/OAEPWithSHA-512AndMGF1Padding" and "AES/CBC/PKCS5PADDING". Other CryLogger false positives are due to non-deterministic test cases in the JulietBench (similar to what we observed during the RVSec experiments).

Conversely, the warnings CryLogger misses (false negatives) in the JulietBench are due to incomplete usage of a cryptographic primitive (CWE-325). Fig. 17 shows an example. In this case, there is a call to the `digest` method without a call to the `update` method of the `MessageDigest` class. CryLogger rules do not address this particular kind of vulnerability, named *Missing Cryptographic Step* (CWE-325) [23]. CryLogger also fails to identify the incorrect initialization of seeds from constant arrays of bytes and the use of the string to keep credential information (CWE-321, CWE-337, CWE-1391). These are the main source of warnings that CryLogger misses in the SmallCryptoAPIBench.

The execution of the CryLogger experiment did not complete for three ApacheCryptoAPIBench projects—leading to 32 false negatives in total. The reason for not completing it varies. In multiple attempts, after executing the CryLogger experiment

```
public void bad() throws Throwable
 {
   if (PRIVATE_STATIC_FINAL_FIVE == 5)
   {
     MessageDigest md = MessageDigest.getInstance("SHA-512");
     /*
        FLAW: Missing call to MessageDigest.update().
        This will result in the hash being of no data
      */
     IO.writeLine(IO.toHex(md.digest()));
   }
 }
```

Fig. 17. JulietBench code where CryLogger misses a vulnerability.

of the Artemis project for less than five minutes, the memory consumption achieves a peak, and the operating system kills the corresponding process Differently, the execution of the CryLogger experiment for Spark and Wicket did not conclude within a time limit of 24 hours. It seems that CryLogger does not scale well for large projects. In particular, including the stack traces to the outputs of CryLogger led to huge log files. Unfortunately, logging the stack traces was necessary to enable us to compare the outcomes of CryLogger with the results of the other tools and the benchmarks' ground truths.

VI. Discussion

In this section we discuss the impact of RVSec on the execution of the ApacheCryptoAPIBench test cases (overhead) (§VI-A), lessons learned and future work (§VI-B), and threats to validity (§VI-C).

A. RV Overhead

Comparing a dynamic analysis approach like Runtime Verification with static analyses necessitates a discussion of the overhead of RV for crypto API misuse detection. More so, using RV to simultaneously monitor many specifications like we do is known to be more costly than monitoring only one specification [54], [55], [56].

Table VII shows the runtime (in seconds) of the tests in ApacheCryptoAPIBench projects without (the "TBase (s)" column) and with (the "TRV (s)" column) RV. It also shows the RVSec overhead (the "Overhead (%)" column). RVSec overhead on these projects ranges from 8.64% (ManifoldCF) to 56.86% (Wicket), with an average of 25.90% and median of 18.32%.

We believe that these RVSec overheads may be acceptable, but they will likely grow as more tests are added to improve coverage. Also, we only measure RVSec overhead on one revision for each project because our focus is on comparing RV with other approaches for detecting crypto API misuses. However, recent evolution-aware techniques were proposed that reduce RV overhead by up to $10x$ (average: $5x$) when running RV across several revisions of a project, e.g., during continuous integration or regression testing [9], [10]. So, using evolution-aware RV to detect crypto API misuses as software evolves could have even lower runtime overheads.

TABLE VII
RVSec Overhead Results for ApacheCryptoAPIBench and Average Running Time of the Static Analyzers

Project	TRV (s)	TBase (s)	Overhead (%)
Dir. Server	21.30	15.00	42.00
Artemis	39.80	35.90	10.86
ManifoldCF	23.90	22.00	8.64
DeltaSpike	47.10	39.80	18.34
Meecrowave	48.40	34.40	40.70
Spark	1,319.70	1,115.40	18.32
Tika	28.00	25.10	11.55
Wicket	24.00	15.30	56.86

B. Lessons Learned and Future Work

Complementary nature of dynamic and static analyses. Our analysis reveals blind spots for RVSec and the static analyzers, CogniCrypt and CryptoGuard. For example, RVSec should be complemented with static analysis tools to find whether a string is hard coded at initialization. Also, static analyzers could benefit from RVSec to reduce false negatives when analyzing sequences of method calls or string manipulation. Lastly, RVSec can help static analyzers to reduce false positive warnings in the presence of configuration files during testing, e.g., by using the recently proposed configuration testing framework, CTests [57].

Recommendations for better usage of RV in crypto API misuse detection. During our research we also identified design recommendations for implementing dynamic analysis tools for detecting crypto API misuses. In particular, we argue that it is important to instrument the client code of the crypto APIs, instead of instrumenting the API code as CryLogger does [22]. First, instrumenting the client code allows a tool to report relevant information of a crypto API misuse (such as where the misuse happens). This kind of information is essential to help a developer understand and fix a misuse. We partially fix this CryLogger limitation by changing some of its components in order to enrich the log files with stack trace information—which substantially increases the log files' size. Second, instrumenting the client code using a JavaMOP-like specification language also makes it easy to consider only misuses that appear in the system under test, or to ignore misuses that happen in some classes (e.g., classes in the Java standard library). Our experience in using CryLogger shows that it does not allow such flexibility. Since our CryLogger extension does not fix this particular issue, we have to remove from the analysis many issues CryLogger reports that appear in the Java standard library.

Issues with existing benchmarks. Several test cases in the SmallCryptoAPIBench contain code that cause runtime exceptions. We detect these problematic code while executing the test cases and implement small fixes to make these tests useful for dynamic analysis. For instance, in ten tests, we increase the size of byte arrays that are used to configure the initialization vectors of ciphers (Fig. 18). This fix is necessary because the initialization vectors require at least 128 bits (16 bytes). SmallCryptoAPIBench also refers to invalid key stores using a URL. We also fix this problem in nine tests (as we show in Fig. 19). Several classes in ApacheCryptoAPIBench needed a

```
1  - byte[] bytes = "abcde".getBytes();
2  + byte[] bytes = "abcde----------".getBytes();
```

Fig. 18. A fix needed to correctly configure initialization vectors on the SmallCryptoAPIBench.

```
1  - cacerts = new URL("https://www.google.com");
2  + cacerts = new File("testInput-ks").toURI().toURL();
```

Fig. 19. A fix needed to correctly explore key stores in the SmallCryptoAPIBench.

fix, though, fortunately, these classes have no use of the JCA library. These fixes are necessary to build the projects and run their tests. We share all these fixes in our replication package. Our experience setting up the OWASPBench and JulietBench was more positive than the other benchmarks.

C. Threats to Validity

We only study the correct usage rules for JCA. So, our results may not generalize to misuse detection in non-JCA crypto APIs. However, future work can evaluate the use of JavaMOP specifications for detecting misuses of other crypto APIs as well. Also, other researchers used RV to find misuses and bugs in non-JCA and non-crypto APIs [11], [12], [13].

Our choice of benchmarks might pose an additional threat to validity. However, the five benchmarks in this article contain a wide variety of JCA usage scenarios. To the best of our knowledge, this is the first study that combines benchmarks curated by researchers (MASCBench, SmallCryptoAPIBench, and MASCBench) and by independent organizations (OWASPBench and JulietBench) for comparing dynamic and static crypto API misuse detectors.

Other than ApacheCryptoAPIBench, the benchmarks were designed for comparing static detectors of crypto API misuses. We re-use these benchmarks almost as is, but we fix several bugs to allow us to execute the programs, which is necessary for RVSec and CryLogger. These benchmarks help to understand the limits of crypto API misuse detectors, but some examples may be fictitious and rare in real-world systems. Indeed, the OWASPBench documentation acknowledges that:

> *The tests are derived from coding patterns observed in real applications, but the majority of them are considerably simpler than real applications.... Although the tests are based on real code, it is possible that some tests may have coding patterns that do not occur frequently in real code.*

So, these benchmarks help us understand the strengths and weaknesses of RVSec, CogniCrypt, CryptoGuard, and CryLogger, but we do not claim that these results would generalize to real systems, except possibly for ApacheCryptoAPIBench.

We revise the ground truth for ApacheCryptoAPIBench, after careful manual analysis of RVSec, CogniCrypt, and CryptoGuard reports. Doing so may add threats to validity, but it improves the benchmark and allows for fairer comparison of RVSec, CogniCrypt, CryptoGuard, and CryLogger—which report crypto API misuses with different levels of detail. We contacted the authors of ApacheCryptoAPIBench, and they agree

```
rule_R01: VIOLATED
rule_R02: VIOLATED
rule_R03: RESPECTED
rule_R04: RESPECTED
rule_R05: RESPECTED
rule_R06: RESPECTED
rule_R07: RESPECTED
rule_R08: RESPECTED
rule_R09: VIOLATED
rule_R10: RESPECTED
...
```

Fig. 20. Snippet of the outcome of the original CryLogger for MASCBench.

```
Violation 01 : rule_R01 : [MessageDigest] algorithm: md5 ::
[java.lang.Thread.getStackTrace,
 java.security.CRYLogger.insertStackTrace,
 java.security.CRYLogger.write,
 java.security.MessageDigest.digest,
 com.minimals.md.differentCase.MD04.main,
 ... ]
...
```

Fig. 21. Snippet of the outcome of our CryLogger implementation for MASCBench.

on the most critical modifications (e.g., our recommendation for treating all uses of `java.util.Random` as secure).

Although many violations (from RVSec, CogniCrypt, CryptoGuard, and CryLogger) are associated with critical CVE/CWE warnings, we do not yet investigate the developers' perceptions of these warnings. Doing so in an in-depth way requires many careful considerations (e.g., adherence to user agreement policies [58] and open-source vulnerability disclosure policies [59]). So, we leave the important work of user validation for future work.

Finally, we re-implemented two core components of CryLogger in order to integrate the tool into our study. Our new implementation was necessary because CryLogger only reports which crypto API rules a system execution violates. Fig. 20 shows a snippet of the outcome of the original CryLogger for the MASCBench. Unfortunately, it is not feasible to compute accuracy metrics using the information present in Fig. 20. Instead, our new CryLogger implementation records all crypto API violations together with the stack trace of the program when a violation happens (see Fig. 21), allowing us to trace the client code that originates the crypto API misuse.

As we mentioned, this new implementation was necessary to integrate CryLogger into our study. Our new implementation generates large log files, which might have led to failing to complete the assessment in three ApacheCryptoAPIBench projects, compromising the CryLogger assessment to this benchmark. Besides that, the original version of CryLogger also does not conclude its execution for the Spark project (one of the ApacheCryptoAPIBench projects). So, even the original CryLogger version might suffer from scalability issues. We contacted the CryLogger authors, asking them for any replication package that could indicate alternatives to compare CryLogger with other tools. Unfortunately, according to the authors, no replication package of previous CryLogger studies is available for sharing.

VII. Related Work

This section discusses work that is most related to ours, including research that focuses on crypto API misuses and the combination of static and dynamic analysis to identify software vulnerability in general.

A. Static Crypto API Misuse Detection

Several static analyses [60] were proposed to assist developers in early detection of vulnerabilities due to crypto API misuses [8], [60], [61], [62], [63], [64]. CogniCrypt [20] and CryptoGuard [7] are two prominent examples of such static analyzers in the literature. CogniCrypt uses rules written in a domain-specific language (DSL) called CrySL to check crypto API usages. CryptoGuard uses optimized slicing-based algorithms to find crypto API misuses. We use the CrySL rules as a basis for developing RVSec specifications because (1) the rules were validated with security experts, (2) the authors provide an extensive test suite that allow us to develop our specifications in a test-driven manner, and (3) the rules are defined as EREs over method call sequences and JavaMOP has native support for ERE as a specification language. As a dynamic analysis-based alternative, we find that RVSec produces fewer false positives and false negatives. So, RVSec can complement static analyses during software development.

B. Dynamic Crypto API Misuse Detection

Dynamic techniques exist for detecting crypto bugs in specific domains. SMV-Hunter [65] and AndroSSL [66] detect SSL/TLS misuses, but they only work for Android. Similarly, iCryptoTracer [67] detects misuses of crypto functions in iOS. K-Hunt [68] finds insecure crypto keys in binaries, so it is not a development-time aid. Unlike these techniques, RV is general and it can be used at development time and across domains—RV was applied to Android [69], [70], [71], cyber-physical systems [72], [73], operating systems [74], [75], and even hardware development [76]. But, RV's generality comes at the cost of writing specifications.

Closer to our study, CryLogger [22] uses 26 rules to detect crypto API misuses. It monitors usage of crypto APIs and logs values of relevant parameters in a file. Then, CryLogger analyzes the logs offline to find violated rules. Differently from CryLogger, RVSec (1) can check inter-class relationships among crypto APIs, (2) monitor the entire life cycle of the instances involved in crypto API usages (and not just values that they use), and (3) pinpoint code locations where RVSec violations occur. RVSec instruments the client code of the APIs, so it generates useful reports for debugging detected crypto API misuses by pinpointing the location of the misuse. Such pinpointing allows for a fair comparison of dynamic and static crypto API misuse detectors. On the other hand, CryLogger only reports that a crypto API misuse was detected, so it was necessary to change the CryLogger implementation to incorporate it into our research.

C. Use of Static and Dynamic Analysis to Detect Other Types of Vulnerability

Combining static and dynamic analysis to identify vulnerability has been explored before. In particular, several research works explore the possible benefits of integrating both program analysis approaches to identify system vulnerability via *anomaly detection*.

For instance, Xu et al. [77] propose a probabilistic reasoning framework that models the program's behavior and context as a joint probability distribution. This distribution captures the dependencies between the program's events and the contextual factors, enabling more accurate anomaly detection. Differently, Shu et al. [78] designed a method for modeling the behavior of a program over a long period and detecting attacks on the program based on the model. The method uses a graph-based representation of the program's behavior, whose nodes represent program states and the edges represent transitions between states. The graph is then analyzed to identify behavior patterns that indicate possible attacks.

Furthermore, Cheng et al. [79] also propose an approach for detecting anomalies in cyber-physical systems (CPS), using event-aware program analysis techniques. The approach first defines a set of events that are expected to occur in a CPS, and then analyzes the program code to identify program states associated with these events. At runtime, the system is monitored for the occurrence of these events, and if they do not occur as expected, it is deemed as an anomaly and flagged for further investigation.

Ahrendt et al. [80] present a different technique to ensure the correctness of software by combining static and runtime verification to check data and control properties of the system. They propose a formalism that models software systems and properties to be verified, allowing for the expression of data and control-related aspects. The framework defines a set of rules that guide the combination of static and runtime techniques, ensuring consistency and coherence in the verification process. Static verification is used to prove that the system satisfies some invariants and preconditions, while runtime verification is used to monitor the system's behavior and detect any violations of post-conditions and temporal properties. The article also introduces a framework and a toolset that support the proposed approach and allow the specification, verification, and monitoring of software systems in a unified way. Differently from our work, their research aims to monitor the systems in production. Instead, our goal with RVSec is to identify crypto API misuses during the execution of test cases. It is a matter of future work to explore techniques that might fix crypto API misuses at runtime.

Handrick et al. [81] present an in-depth analysis related to the combination of static and dynamic analysis to identify Android malware—using the *Android Mining Sandbox Approach* [82], [83]. The authors bring evidence that static and dynamic analysis complement each other, improving the overall accuracy of Android malware classification. Our work differs from this literature because we target crypto API misuses, a specific and pressing source of software vulnerability. However, we believe the related work we presented in this section might give possible

directions for integrating static and dynamic analysis tools that detect crypto API misuses.

VIII. Conclusions

We evaluate the use of RV for detecting crypto API misuses. To do so, we implemented RVSec after translating specifications CrySL rules [6], [20] into JavaMOP specs [32] and use RVSec to identify misuses of the Java Cryptographic Architecture (JCA) API. We compare the accuracy of RVSec with the state-of-the-art tools CogniCrypt [6], [20], [84], CryptoGuard [7], and CryLogger [22] on five benchmarks (three from the literature and two from independent organizations).

The results show that, on average, the accuracy (F_1 score) of RVSec (0.95) is higher than the accuracy of CogniCrypt (0.83), CryptoGuard (0.78), and CryLogger (0.86). We analyze the strengths and weaknesses of RVSec, CogniCrypt, CryptoGuard, and CryLogger and provide evidence that static and dynamic analyses can be complementary for identifying crypto API misuses. Our work resulted in fixes to CogniCrypt, leading to an improvement in its precision, and also to the benchmarks that are commonly used to evaluate crypto API misuse detectors.

In the future, we plan to run RVSec in Android apps. Additional engineering effort is required to run JavaMOP on Android. We also plan to explore model-based test generation of JavaMOP specifications to augment the ability of existing test suites to catch bugs.

References

[1] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do Java developers struggle with cryptography APIS?" in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: ACM, 2016, pp. 935–946, doi: 10.1145/2884781.2884790.

[2] E. Barker and A. Roginsky, "Transitioning the use of cryptographic algorithms and key lengths," 2019. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-131Ar2.

[3] "Cryptographic mechanisms: Recommendations and key lengths," German Federal Office for Information Security, Bonn, Germany, Tech. Rep. BSI TR-02102-1, 2022. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile.

[4] Y. Acar et al., "Comparing the usability of cryptographic APIS," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2017, pp. 154–171.

[5] F. Fischer et al., "Stack overflow considered harmful? The impact of copy & paste on android application security," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2017, pp. 121–136.

[6] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An extensible approach to validating the correct usage of cryptographic APIs," in *Proc. 32nd Eur. Conf. Object-Oriented Program. (ECOOP)*, T. Millstein, Ed., vol. 109. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 10:1–10:27.

[7] S. Rahaman et al., "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: ACM, 2019, pp. 2455–2472, doi: 10.1145/3319535.3345659.

[8] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: ACM, 2013, pp. 73–84, doi: 10.1145/2508859.2516693.

[9] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Rosu, and D. Marinov, "Techniques for evolution-aware runtime verification," in *Proc. 12th IEEE Conf. Softw. Testing, Validation Verification (ICST)*, 2019, pp. 300–311.

[10] O. Legunsen, D. Marinov, and G. Rosu, "Evolution-aware monitoring-oriented programming," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 2, 2015, pp. 615–618.

[11] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2016, pp. 602–613.

[12] O. Legunsen, N. A. Awar, X. Xu, W. U. Hassan, G. Rosu, and D. Marinov, "How effective are existing Java API specifications for finding bugs during runtime verification?" *Autom. Softw. Eng.*, vol. 26, no. 4, pp. 795–837, 2019, doi: 10.1007/s10515-019-00267-1.

[13] B. Miranda, I. Lima, O. Legunsen, and M. d'Amorim, "Prioritizing runtime verification violations," in *Proc. IEEE 13th Int. Conf. Softw. Testing, Validation Verification (ICST)*, 2020, pp. 297–308.

[14] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE),* L. K. Dillon, W. Visser, and L. A. Williams, Eds. Austin, TX, USA: ACM, May 14–22, 2016, pp. 37–48, doi: 10.1145/2884781.2884782.

[15] Q. Luo et al., "RV-monitor: Efficient parametric runtime verification with simultaneous properties," in *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds. Cham, Switzerland: Springer International Publishing, 2014, pp. 285–300.

[16] C. Lee, D. Jin, P. O. Meredith, and G. Roşu, "Towards categorizing and formalizing the JDK API," Comput. Sci. Dept., Univ. Illinois Urbana-Champaign (UIUC), Tech. Rep., 2012.

[17] L. Teixeira, B. Miranda, H. Rebêlo, and M. d'Amorim, "Demystifying the challenges of formally specifying API properties for runtime verification," in *Proc. 14th IEEE Conf. Softw. Testing, Verification Validation (ICST)*, 2021, pp. 82–93.

[18] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Softw. Eng.*, vol. 39, no. 5, pp. 613–637, 2013.

[19] M. Gabel and Z. Su, "Testing mined specifications," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng. (FSE)*. New York, NY, USA: ACM, 2012, pp. 1–11.

[20] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CRYSL: An extensible approach to validating the correct usage of cryptographic APIS," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2382–2400, Nov. 2021.

[21] S. Afrose, Y. Xiao, S. Rahaman, B. Miller, and D. D. Yao, "Evaluation of static vulnerability detection tools with Java cryptographic API benchmarks," *IEEE Trans. Softw. Eng.*, vol. 49, no. 2, pp. 485–497, Feb. 2023.

[22] L. Piccolboni, G. D. Guglielmo, L. P. Carloni, and S. Sethumadhavan, "CRYLOGGER: Detecting crypto misuses dynamically," in *Proc. 42nd IEEE Symp. Secur. Privacy (SP),* San Francisco, CA, USA. Piscataway, NJ, USA: IEEE, 2021, pp. 1972–1989, doi: 10.1109/SP40001.2021.00010.

[23] "Missing cryptographic step," MITRE, CWE-ID CWE-325. 2006. Accessed: Apr. 23, 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/325.html

[24] A. S. Ami, N. Cooper, K. Kafle, K. Moran, D. Poshyvanyk, and A. Nadkarni, "Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2022, pp. 614–631.

[25] "OWASP benchmark." OWASP. Accessed: Jun. 16, 2022. [Online]. Available: https://owasp.org/www-project-benchmark

[26] National Security Agency (NSA), "Juliet benchmark," 2017. Accessed: Sep. 1, 2023. [Online]. Available: https://samate.nist.gov/SARD/test-suites/111

[27] D. Hook, *Beginning Cryptography With Java*, 1st ed. Birmingham, U.K.: Wrox Press, 2005.

[28] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering: Design Principles and Practical Applications*. Hoboken, NJ, USA: Wiley, 2010.

[29] "Use of a broken or risky cryptographic algorithm," MITRE, CWE-ID CWE-327. Accessed: Dec. 10, 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/327.html

[30] *Java Cryptography Architecture (JCA) Reference Guide.* (2022). Java Platform. Accessed: Dec. 10, 2022. [Online]. Available: https://docs.oracle.com/en/java/javase/11/security/index.html

[31] S. Krüger, "CogniCrypt – The secure integration of cryptographic software," Ph.D. dissertation, Universität Paderborn, Paderborn, Germany, May 2020. Accessed: Dec. 10, 2022. [Online]. Available: https://www.bodden.de/pubs/phdKrueger.pdf

[32] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, 2012, pp. 1427–1430.

[33] "JavaMOP." GitHub. Accessed: Jan. 20, 2022. [Online]. Available: https://github.com/runtimeverification/javamop

[34] F. Chen and G. Roşu, "Parametric trace slicing and monitoring," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer, 2009, pp. 246–261.

[35] F. Chen, P. O. Meredith, D. Jin, and G. Rosu, "Efficient formalism-independent monitoring of parametric properties," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 383–394.

[36] "Apache QPID Broker-J." QPID APACHE. Accessed: Dec. 10, 2022. [Online]. Available: https://qpid.apache.org/components/broker-j/index.html

[37] National Security Agency (NSA), "Juliet test suite for Java (User Guide)," Center for Assured Software, Nat. Inst. Stand. Technol. (NIST), Gaithersburg, MD, USA, Tech. Rep., 2012.

[38] S. Afrose, S. Rahaman, and D. Yao, "CryptoAPI-bench: A comprehensive benchmark on Java cryptographic API misuses," in *Proc. IEEE Cybersecurity Develop. (SecDev)*, 2019, pp. 49–61.

[39] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. D. Yao, and N. Meng, "Example-based vulnerability detection and repair in Java code," in *Proc. 30th IEEE/ACM Int. Conf. Program Comprehension (ICPC)*. New York, NY, USA: ACM, 2022, pp. 190–201, doi: 10.1145/3524610.3527895.

[40] P. Ferrara, E. Burato, and F. Spoto, "Security analysis of the OWASP benchmark with Julia," in *Proc. 1st Italian Conf. Cybersecurity (ITASEC)*, A. Armando, R. Baldoni, and R. Focardi, Eds., vol. 1816. Venice, Italy: CEUR-WS, Jan. 17–20, 2017, pp. 242–247. [Online]. Available: http://ceur-ws.org/Vol-1816/paper-24.pdf

[41] B. Mburano and W. Si, "Evaluation of web vulnerability scanners based on OWASP benchmark," in *Proc. 26th Int. Conf. Syst. Eng. (ICSEng)*. Piscataway, NJ, USA: IEEE, 2018, pp. 1–6.

[42] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. D. Yao, and N. Meng, "Automatic detection of Java cryptographic API misuses: Are we there yet?" *IEEE Trans. Softw. Eng.*, vol. 49, no. 1, pp. 288–303, Jan. 2023.

[43] D. Beyer, "Software verification: 10th comparative evaluation (SV-Comp 2021)," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. F. Groote and K. G. Larsen, Eds. Cham, Switzerland: Springer International Publishing, 2021, pp. 401–422.

[44] J. Herter, D. Kästner, C. Mallon, and R. Wilhelm, "Benchmarking static code analyzers," *Rel. Eng. Syst. Saf.*, vol. 188, pp. 336–346, Aug. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0951832018304721

[45] "JaCoCo Code Coverage." Accessed: Jun. 15, 2022. [Online]. Available: https://www.eclemma.org/jacoco/

[46] "CrySL repository." GitHub. Accessed: Jan. 20, 2022. [Online]. Available: https://github.com/CROSSINGTUD/Crypto-API-Rules/

[47] M. Hazhirpasand, M. Ghafari, and O. Nierstrasz, "Java cryptography uses in the wild," in *Proc. 14th ACM / IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*. New York, NY, USA: ACM, 2020, doi: 10.1145/3382494.3422166.

[48] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *Proc. IEEE Int. Conf. Softw. Testing, Verification Validation (ICST)*, 2016, pp. 80–90.

[49] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, 2018, pp. 433–444.

[50] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! Detecting flaky tests caused by concurrency with shaker," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2020, pp. 301–311.

[51] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proc. 17th Int. Conf. Mining Softw. Repositories (MSR)*. New York, NY, USA: ACM, 2020, pp. 492–502, doi: 10.1145/3379597.3387482.

[52] "Use of hard-coded credentials," MITRE, CWE-ID CWE-798. Accessed: Dec. 10, 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/798.html

[53] S. Arzt et al., "FLOWDROID: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, M. F. P. O'Boyle and K. Pingali, Eds. Edinburgh, U.K.: ACM, Jun. 9–11, 2014, pp. 259–269, doi: 10.1145/2594291.2594299.

[54] D. Jin, P. O. Meredith, and G. Roşu, "Scalable parametric runtime monitoring," Univ. Illinois Urbana-Champaign (UIUC), Urbana, IL, USA, Tech. Rep., 2012.

[55] P. Meredith and G. Roşu, "Efficient parametric runtime verification with deterministic string rewriting," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2013, pp. 70–80.

[56] Q. Luo et al., "RV-monitor: Efficient parametric runtime verification with simultaneous properties," in *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds. Cham, Switzerland: Springer International Publishing, 2014, pp. 285–300.

[57] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing configuration changes in context to prevent production failures," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation (OSDI 20)*. Berkeley, CA, USA: USENIX Association, Nov. 2020, pp. 735–751. Accessed: Dec. 10, 2022. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/sun

[58] "ACM publications policy on research involving human participants and subjects," 2002. Accessed: Dec. 10, 2022. [Online]. Available: https://www.acm.org/publications/policies/research-involving-human-participants-and-subjects

[59] B. Carlson, K. Leach, D. Marinov, M. Nagappan, and A. Prakash, "Open source vulnerability notification," in *Proc. IFIP Int. Conf. Open Source Syst.* Cham, Germany: Springer, 2019, pp. 12–23.

[60] L. Sampaio and A. Garcia, "Exploring context-sensitive data flow analysis for early vulnerability detection," *J. Syst. Softw.*, vol. 113, pp. 337–361, Mar. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215002873

[61] P. Arteau. "Find security bugs." GiHub. Accessed: Jun. 15, 2022. [Online]. Available: https://find-sec-bugs.github.io/

[62] "Xanitizer." RigsIT. Accessed: Jun. 15, 2022. [Online]. Available: https://www.rigs-it.net

[63] "SonarQube." SonarSource. Accessed: Jun. 15, 2022. [Online]. Available: https://www.sonarqube.org/

[64] NCCGroup. "Visualcodegrepper." GitHub. Accessed: Jun. 15, 2022. [Online]. Available: https://github.com/nccgroup/VCG

[65] D. S. J. S. G. Greenwood and Z. L. L. Khan, "SMV-hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in android apps," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS) Internet Soc.*, San Diego, CA, USA. Reston, VA, USA: Citeseer, 2014, pp. 1–14.

[66] F. Gagnon, M.-A. Ferland, M.-A. Fortier, S. Desloges, J. Ouellet, and C. Boileau, "AndroSSL: A platform to test android applications connection security," in *Proc. Int. Symp. Found. Pract. Secur*. Cham, Germany: Springer, 2015, pp. 294–302.

[67] Y. Li, Y. Zhang, J. Li, and D. Gu, "ICRYPTOTRACER: Dynamic analysis on misuse of cryptography functions in IoS applications," in *Proc. Int. Conf. Netw. Syst. Secur.* Cham, Germany: Springer, 2015, pp. 349–362.

[68] J. Li, Z. Lin, J. Caballero, Y. Zhang, and D. Gu, "K-Hunt: Pinpointing insecure cryptographic keys from execution traces," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 412–425.

[69] Y. Falcone, S. Currea, and M. Jaber, "Runtime verification and enforcement for android applications with RV-droid," in *Proc. Int. Conf. Runtime Verification*. Cham, Germany: Springer, 2012, pp. 88–95.

[70] A. Bauer, J.-C. Küster, and G. Vegliach, "Runtime verification meets android security," in *NASA Formal Methods Symp*. Cham, Germany: Springer, 2012, pp. 174–180.

[71] P. Daian et al., "RV-android: Efficient parametric android runtime verification, a brief tutorial," in *Runtime Verification*. Cham, Germany: Springer, 2015, pp. 342–357.

[72] X. Zheng, C. Julien, R. Podorozhny, F. Cassez, and T. Rakotoarivelo, "Efficient and scalable runtime monitoring for cyber–physical system," *IEEE Syst. J.*, vol. 12, no. 2, pp. 1667–1678, Jun. 2018.

[73] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez, "BraceAssertion: Runtime verification of cyber-physical systems," in *Proc. IEEE 12th Int. Conf. Mobile Ad Hoc Sensor Syst.* Piscataway, NJ, USA: IEEE, 2015, pp. 298–306.

[74] J. Huang et al., "ROSRV: Runtime verification for robots," in *Proc. Int. Conf. Runtime Verification*. Cham, Germany: Springer, 2014, pp. 247–254.

[75] D. B. d. Oliveira, T. Cucinotta, and R. S. d. Oliveira, "Efficient formal verification for the Linux kernel," in *Proc. Int. Conf. Softw. Eng. Formal Methods*. Cham, Germany: Springer, 2019, pp. 315–332.

[76] D. Solet, J.-L. Béchennec, M. Briday, S. Faucou, and S. Pillement, "Hardware runtime verification of embedded software in SOPC," in *Proc. 11th IEEE Symp. Ind. Embedded Syst. (SIES)*, 2016, pp. 1–6.

[77] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection

with context sensitivity," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*. Piscataway, NJ, USA: IEEE, 2016, pp. 467–478.

[78] X. Shu, D. Yao, N. Ramakrishnan, and T. Jaeger, "Long-span program behavior modeling and attack detection," *ACM Trans. Privacy Secur.*, vol. 20, no. 4, pp. 1–28, 2017.

[79] L. Cheng, K. Tian, D. D. Yao, L. Sha, and R. A. Beyah, "Checking is believing: Event-aware program anomaly detection in cyber-physical systems," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 2, pp. 825–842, Mar./Apr. 2021.

[80] W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider, "Verifying data-and control-oriented properties combining static and runtime verification: Theory and tools," in *Formal Methods in System Design*, vol. 51. Cham, Germany: Springer, 2017, pp. 200–265.

[81] F. H. da Costa et al., "Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification," *J. Syst. Softw.*, vol. 183, Jan. 2022, Art. no. 111092, doi: 10.1016/j.jss.2021.111092.

[82] L. Bao, T. B. Le, and D. Lo, "Mining sandboxes: Are we there yet?" in *Proc. 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. Campobasso, Italy: IEEE Computer Society, Mar. 20–23, 2018, pp. 445–455, doi: 10.1109/SANER.2018.8330231.

[83] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. Austin, TX, USA: ACM, May 14–22, 2016, pp. 37–48, doi: 10.1145/2884781.2884782.

[84] S. Krüger et al., "CogniCrypt: Supporting developers in using cryptography," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. Urbana, IL, USA: IEEE Computer Society, Oct. 30–Nov. 3, 2017, pp. 931–936, doi: 10.1109/ASE.2017.8115707.