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—Cryptographic (crypto) API misuses often cause security vulnerabilities, so static and dynamic analyzers were recently proposed to detect such misuses. These analyzers differ in strengths and weaknesses, and they can miss bugs. Motivated by the inherent limitations of existing analyzers, we study runtime verification (RV) as an alternative for crypto API misuse detection. RV monitors program runs against formal specifications and 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. 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 show that RV is effective for detecting crypto API misuses and highlight the strengths and limitations of these tools. We also discuss how static and dynamic analysis can complement each other for detecting crypto API misuses.

1 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 paper 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 and dynamic analyzers are quite effective in detecting crypto API misuses [6], [7], [8], [9], we show that they are limited for detecting some 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 paper, we study the use of runtime verification (RV) as an alternative for detecting crypto API misuses. We also compare RV's accuracy with those of state-of-the-art static and dynamic analyzers.

- *A. Torres, P. Costa, L. Amaral, J. Pastro, R. Bonifácio, and E. D. Canedo are with the Computer Science Department, University of Brasília, Brasília, Brazil. E-mail: {adriano.torres,teixeira.pedro,luis.amaral,jonata.pastro}@aluno.unb.br and {rbonifacio,ednacanedo}@unb.br.*
- *M. d'Amorim is with the Department of Computer Science, North Carolina State University, Raleigh, USA, and the Informatics Center, Federal University of Pernambuco, Recife, Brazil. E-mail: mdamori@ncsu.edu.*
- *O. Legunsen is with the Department of Computer Science, Cornell University, Ithaca, USA. E-mail: legunsen@cornell.edu.*
- *E. Bodden is with the Paderborn University and the Fraunhofer Institute for Mechatronic Systems Design, Paderborn, Germany. E-mail: eric.bodden@uni-paderborn.de.*

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

The inputs to RV 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 a program execution violates any specifications. The RV tool that we use in this paper is based on JavaMOP [10].

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 [11], [12], [13], [14], [15]. Second, RV can complement static analysis—which has good coverage but often *over-approximates the program behavior*, leading to false alarms [16]. Instead, a bug-free RV tool with perfect specifications generates no false positives, but it 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 the runtime overhead that RV incurs. 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 each program, but, in practice, techniques exist that can be used to significantly speed up RV during software evolution [11], [12].

Evaluating the accuracy of RV is particularly challenging because manually writing specifications is notoriously difficult [17], [18], [19]. In particular, translating crypto recommendations, informally available in crypto standards, Common Vulnerabilities and Exposures (CVEs), and Common Weakness Enumerations (CWEs), for a specific crypto API and RV implementation requires a thorough validation process. Moreover, although specification miners exist [20], they are often imprecise [21] and can infer API misuses as

specifications [22]. Unfortunately, our research also reveals that existing datasets of crypto API *uses* and *misuses* in the literature [23] 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 in existing datasets may have led to inflated performance numbers of some static analyzers in prior work.

To obtain RV specifications, we use 22 CrySL rules [6], [22] that were validated by independent security experts. We manually translate these CrySL rules into 22 JavaMOP specifications. JavaMOP is a natural choice for our investigation, 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 method-call order*.

We were unable to check ordering constraints using CryLogger [9], a recently proposed dynamic analysis for detecting crypto API misuses. CryLogger relies on API code instrumentation, instead of client code instrumentation, yielding crypto API warnings that are hard to use in empirical assessments. Also, unlike our JavaMOP-based approach, CryLogger cannot check ordering constraints on method calls. This latter limitation makes CryLogger unable to detect some crypto API misuses, e.g., the missing cryptographic step in CWE 325 [24].

To obtain ground truth for comparison, we adapt five publicly available Java benchmarks [23], [25], [26], [27] 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, more than 350K lines of code (LOC), and they are a mix of small Java programs (14–52 LOC) and open-source programs (between 6,788 and 164,335 LOC). Some 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 analyzers—CogniCrypt, CryptoGuard, and CryLogger—on these benchmarks. We also evaluate the runtime overhead of RVSec and the impact of code coverage on its accuracy. To compare RVSec with CryLogger, we extend the CryLogger implementation to log the stack traces whenever client code makes calls to methods in a crypto library. We perform quantitative and qualitative comparison of both dynamic approaches—RVSec and CryLogger.

Overall, RVSec achieves an average F_1 measure of 95%, compared with 83%, 78%, and 86% for CogniCrypt, CryptoGuard, and CryLogger, respectively (see §4, Table 3). On a larger benchmark, RVSec overhead varies between 8.64% and 56.86% (see §6.1, Table 7). Note that we did not apply recent optimizations that speed up RV during software evolution [11], [12], [28].

Our qualitative analysis shows that RV and static analyzers are complementary. We found crypto API misuses that only RVSec detected. Also, some crypto API misuses were detected by static analyzers but not RVSec.

We make the following contributions:

```
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 [23].

(a) An in-depth study that compares RVSec with state-of-the-art tools for detecting crypto API misuses (CogniCrypt, CryptoGuard, and CryLogger). We discuss strengths and weaknesses of these tools with respect to their crypto API misuse detection capabilities.
(b) We find that static and dynamic analyses are complementary for crypto API misuse identification. Specifically, 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.

2 Example

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

An example crypto API misuse. Figure 1 shows a crypto API misuse from the Apache project, Meecrowave [23]. The code snippet encrypts data using the `Cipher` class from the Java Cryptography Architecture (JCA) API [29]. To do so, a `Cipher` is initialized with the `DESede` algorithm, which implements the weaker Triple DES Encryption algorithm [30] that should no longer be used in production [2], [31].

To correctly use a `Cipher`, developers should (1) use a secure cryptographic configuration to obtain a `Cipher` instance (Line 3); (2) initialize the `Cipher` object using a `key` that is consistent with the `Cipher` algorithm being used (Line 4); and (3) use a specified order of method calls (lines 3-5). Departure from these conditions on `Cipher` can result in security vulnerabilities [29], [32], [33].

Static detection of crypto API misuses. Simple static analyses (or even `grep`) can detect the crypto API misuse on line 3 in Figure 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 the order of method calls is valid and that certain values do not propagate to sensitive locations.

To make it easier to detect crypto API misues, static analyses were recently proposed, e.g., CogniCrypt [22] and CryptoGuard [7]. Unfortunately, even these advanced static analysis tools can fail, producing false positives or false negatives as a result. For instance, in the code snippet in

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

```
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()`.

Figure 2, CryptoGuard raises a "Found broken hash function" warning in method `withMD5()`. However, this warning is a false positive; `main()` does not call this `withMD5()` method that configures the `MDHelper` instance using the non-recommended `MD5` hash algorithm. CogniCrypt does not raise any alarm, though, even if `MDHelper` is used with a call to `MDHelper.instance().withMD5().digest(str)` (not shown in Figure 2). That is, CogniCrypt produces a false negative.

Dynamic detection of crypto API misuses. We investigate runtime verification (RV) as a dynamic-analysis alternative for detecting crypto API misuses and compares the resulting approach, RVSec, with state-of-the-art crypto API misuse detectors: 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*, like method calls or field accesses, against the specification. Monitors are typically automata and the event sequences are called *traces*. We use JavaMOP [10], [17], [34] as the RV tool in this paper; it was used to find hundreds of bugs during unit testing of many open-source projects [13], [14], [15].

Figure 3 shows an RV specification in the JavaMOP syntax; we wrote it based on the CrySL specification for the JCA `Cipher` class. (CrySL is CogniCrypt's specification language [6], [22]). Figure 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 matches the specification. The Extended Regular Expression (ERE) on line 15, Figure 3, defines the property; it matches traces where events `g1` or `g2` occur exactly once, followed by one or more `init` events, and exactly one `doFinal` event. We use EREs to write most JavaMOP specifications; CrySL specs also use the ERE formalism. But, JavaMOP can monitor properties written in other formalisms, like Context Free Grammar (CFG), Finite State Machine (FSM), or Linear Temporal Logic (LTL). In a few CrySL specifications that involved many events, we used the FSM formalism for ease of understanding.

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 re-initialized. JavaMOP specifications are *parametric* [35], [36]. So, one monitor will be synthesized for every unique set of specification parameters. Since `Cipher` is the only specification parameter (line 1, Figure 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 Figure 1, RVSec has better precision and recall when analyzing the code in Figure 2. We also compare RVSec with CryLogger, a dynamic analysis for detecting crypto API misuses.

3 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.

We modified CryLogger to enable our experiments and fair comparison. CryLogger has two main components: an implementation of JCA classes that logs API usage information, and a Python module that processes the log files and outputs crypto API misuses. We modified both components. First, we changed the implementation of the JCA classes to also log the JVM stack trace at the crypto API call sites. Second, we changed the Python module to also output client methods from which calls to crypto APIs originate.

3.1 Benchmarks

Table 1 shows the five benchmarks that we evaluate:

(a) **MASCBench** contains 30 small Java programs containing crypto API misuses [25]. These programs are curated from open-source Android apps and Apache Qpid Brokerj [37] and minimized to only the crypto API misuses.

```
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 1
Benchmarks in our study.

Benchmark	Used In	TCs	SLOC	Misuses
MASCBench	[25]	30	0.5K	28
SmallCryptoAPIBench	[23], [39], [40]	187	3.7K	130
OWASPBench	[41], [42], [43]	482	47.5K	259
JulietBench	[44], [45]	102	6.8K	102
ApacheCryptoAPIBench	[23], [39], [40]	-	303.3K	74

TABLE 2
Details of the ApacheCryptoAPIBench, highlighted in Table 1.

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

(b) **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. [23] benchmark.

(c) **OWASPBench** contains 482 test cases related to crypto APIs. The test suite is curated by OWASP (https://owasp.org) [26]. 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).

(d) **JulietBench** contains 102 test cases curated by the US National Security Agency (NSA) [27], [38]. The breakdown of tests per CWE is: 17 CWE-325 (Missing Cryptographic Step), 34 CWE-327, and 51 CWE-328.

(e) **ApacheCryptoAPIBench** contains crypto API misuses from Apache projects [23] involving eight Apache projects with varying test coverage. Those projects have 3169 test cases, but some are not related to JCA. Table 2 shows more details about ApacheCryptoAPIBench. CryLogger times out on three ApacheCryptoAPIBench projects: Artemis, Spark, and Tika.

3.2 Procedure and Metrics

To evaluate accuracy, we compute Precision, Recall, and F_1 score, metrics typically used in related work [23], [40]:

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (3)$$

Besides accuracy, we evaluate RVSec's overhead using only the open-source projects in ApacheCryptoAPIBench, which allow for realistic assessment. To measure RVSec overhead, we run ApacheCryptoAPIBench projects' tests ten times with and without RV, and average the execution times. TRV and $TBase$ are average times to run test cases with and without RV, respectively. RV overhead corresponds to the percentage time increase of TRV over $TBase$, computed as $100 \times (TRV - TBase)/TBase$. Lastly, we measure statement, branch, and method coverage in ApacheCryptoAPIBench using JaCoCo [46], to investigate how coverage impacts RVSec's accuracy and overhead.

3.3 RV of the JCA Crypto API using JavaMOP

We write JavaMOP JCA specifications, such as the one in Figure 3 (explained in §2), to check for crypto API misuses. To do so, we start from an existing set of CrySL JCA specifications for three main reasons. First, these CrySL specifications were validated by crypto experts. Second, the CrySL and JavaMOP specification languages are similar. Third, the CogniCrypt development team provide a test suite (with 31 test classes and over 200 test methods) that allow us to more easily validate our JavaMOP specifications.

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

The CrySL repository [47] contains rules for 47 JCA classes [22]. But, previous studies show that a subset of 12 JCA classes (including `MessageDigest`, `Cipher`, `Signature`,

TABLE 3
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

and `KeyGenerator`) is more frequently used and yields most crypto API misuses [22], [48]. So, we start writing JavaMOP specifications for these 12 classes and end up with ten more JavaMOP specifications for classes that they depend on. In total, we write 22 JavaMOP specifications used in our study.

4 Accuracy of the Evaluated Techniques

We present quantitative results from comparing the accuracy of RVSec with those of other crypto API misuse detectors: CogniCrypt, CryptoGuard, and CryLogger. Table 3 shows the result of each tool on the five benchmarks.

MASCBench

For MASCBench, all four tools have 100% Precision; the "0s" in the "FP" rows indicate that no tool reports a false positive. RVSec also does not miss any crypto API misuse (so, a Recall of 100%), while CogniCrypt, CryptoGuard, and CryLogger miss five, nine, and eight misuses, respectively.

The false negatives in CogniCrypt and CryptoGuard are due to limitations in the static analyses that they implement. For instance, if a call to `keyGenerator.getInstance()` returns the unsafe crypto schema, "AES" (mapped to the schema "AES/ECB/PKCS5Padding"), CogniCrypt and CryptoGuard fail to identify the crypto API misuse in `Cipher.getInstance(keyGenerator.getInstance());`.

False negatives in CryLogger occur because the tool fails to detect when an initialization vector parameter specification is initialized using a constant byte array (for four CryLogger false negatives, see an example in Figure 4). Three other CryLogger false negatives are due to test cases that incorrectly seed secure random values (see an example in Figure 5). We provide more details in §5.

```
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.

```
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 incorrect seeding of random values.

SmallCryptoAPIBench

RVSec does not achieve maximum accuracy (i.e., F_1=1.0) in this benchmark, but it achieves higher accuracy than the other tools. Seven (of eight) misuses that RVSec misses are related to usage of hard-coded passwords for loading key stores (e.g., lines 3 and 6 in Figure 6). Failure to handle hard-coded strings is a limitation of RVSec—it is hard to check at runtime whether a variable has been initialized to a hard-coded string constant.

```
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.

The CryLogger false positives in the SmallCryptoAPIBench are due to overly strong constraints. For example, CryLogger signals warnings even for safe `Cipher` crypto-schemes (a ALGORITHM/MODE/PADDING string such as `AES/CBC/PKCS5Padding`.). Further, CryLogger generated the highest number of false negatives in SmallCryptoAPIBench, mainly because it does not identify the use of predictable seeds and credentials in strings. We provide more examples, and further analyze the false positives and false negatives from all tools in §5.

OWASPBench

OWASPBench is the evaluated benchmark with the highest number of test cases that involve API misuses (Table 1). RVSec again achieves the highest accuracy (F_1 score = 0.99) in OWASPBench, compared to the other tools. RVSec, CogniCrypt, and CryLogger did not miss any crypto API misuse. But, 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 201 cases. Differently, CryptoGuard reports 27 false positives, all related to loading a crypto schema from a configuration file. Figure 7 shows an illustrative code snippet.

```
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.

At runtime, the call to `getProperty()` in Figure 7 retrieves the valid crypto schema `AES/GCM/NoPadding`, but CryptoGuard wrongly approximates that a different value, `AES/ECB/PKCS5Padding`, is assigned. Similarly, CryptoGuard assumes valid algorithms that appear as alternatives to invalid ones in a 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. But, OWASPBench uses a configuration file to assign an unsafe hashing algorithm, `MD5`, to the string `"hashAlg1"` as shown in Figure 8. We observe that configuration file usage is a major source of CryptoGuard imprecision, causing many false positives and false negatives in OWASPBench.

```
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.

CryLogger yields no false negative in OWASPBench (recall of 100%), but it generates 53 false positives. Our manual analysis reveals that these false positives are also caused by overly strong constraints on the CryLogger rules that raise unnecessary warnings in safe crypto schemes.

JulietBench

RVSec achieves higher F_1 score than static tools on JulietBench. For Recall, RVSec misses two misuses, CryptoGuard misses 17 misuses, CryLogger misses 16, and CogniCrypt does not miss any. Our manual analysis shows that both 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 [49], [50], [51], [52]. Such flaky-test mitigation techniques should be investigated in the future to reduce the impact of non-deterministic runs on 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, which instantiates a byte array using `getEncoded()` from the `SecretKey` class:

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

All false positives that CogniCrypt reports for JulietBench are due to this idiom.

ApacheCryptoAPIBench

Unfortunately, the provided ground truth in ApacheCryptoAPIBench was missing essential information that we need to compute Precision and Recall. For example, many *true positives* in the original ApacheCryptoAPIBench ground truth do not specify the Java class in which the related crypto API misuse occurs. We also found and shared with the authors of ApacheCryptoAPIBench cases where the labels assigned to warnings in the ground truth dataset were incorrect. They agreed with almost all our observations and changed their benchmark.

Despite these issues with the original ground truth, the ApacheCryptoAPIBench benchmark can still be a valuable source of data on how RVSec compares with other tools on real open-source projects on which the static detectors were previously evaluated. So, we manually inspect all warnings generated by RVSec, CogniCrypt, and CryptoGuard on ApacheCryptoAPIBench to generate our ground truth. We count all unique warnings from a class/method. This decision is needed because the tools may generate different numbers of warnings for a unique misuse.

We obtain 192 unique warnings from ApacheCryptoAPIBench. Of these, we remove (a) 44 warnings from third-party libraries, and (b) 14 RVSec warnings from test classes (for fair comparison with CogniCrypt and CryptoGuard, which do not analyze test classes). Our final dataset of 134 warnings is summarized in Table 4.

Creating our own ground truth may introduce threats to validity, but doing so allows us to avoid inconsistencies that we found in the original ApacheCryptoAPIBench's ground truth. We do not use CryLogger's outputs to build our ApacheCryptoAPIBench ground truth for two reasons. First, the CryLogger finished for only five of eight ApacheCryptoAPIBench projects. Second, CryLogger generates thou-

TABLE 4
Summary of warnings techniques report on ApacheCryptoAPIBench.

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

sands of warnings for ApacheCryptoAPIBench, many of which are in standard Java classes, not in the application. It is not feasible to manually validate all these warnings.

Table 3 shows the Precision, Recall, and F_1 score based on the ground truth dataset that we curate for ApacheCryptoAPIBench. RVSec has the highest Precision, but CogniCrypt has a higher F_1 score. CryLogger has the highest number of false negatives (34) because of the projects on which it times out. Also, the eight false positives that CryLogger reports is somewhat high. There would be more false positives if CryLogger finished executing all projects. We do not summarize these results in Table 3, to avoid confusion.

```
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.

CryptoGuard finds only 58% of ApacheCryptoAPIBench misuses because it implements fewer rules than CogniCrypt. RVSec misses 12 of 48 misuses that CogniCrypt detects. We find that these false negatives are due to lack of (a) test cases to reveal the misuses, and (b) RV specifications for infrequently used JCA classes (e.g., `SecretKeyFactory` and `TrustManagerFactory`). Future work should write specifications for these classes.

RVSec reports one false positive, while CogniCrypt, CryptoGuard, and CryLogger report 10, 14 and 13, respectively. Thirteen false positives from CryptoGuard result from a rule that approximates usages of `java.util.Random` to be insecure. But, `java.util.Random` is often used in *non*-cryptographic contexts. We manually find no usage of the `java.util.Random` class in the ApacheCryptoAPIBench that is vulnerable. We confirm these cases with ApacheCryptoAPIBench authors [23], and mark them as false positives.

If these `java.util.Random` usages were true positives, CryptoGuard Precision on ApacheCryptoAPIBench would be 0.95. We keep these warnings in our dataset because previous research wrongly labeled them as true positives, causing misleading results to be published [23], [40]. These safe usages of `java.util.Random` are also the main source of CryLogger false positives in ApacheCryptoAPIBench.

The 10 false positives that CogniCrypt reports involve tricky situations. For example, in Tika, code that prepares a `Cipher` may not explicitly call methods like `update()` or `doFinal()`, as in Figure 9. CogniCrypt reports a misuse, but we find that developers expect API clients to make those missing calls at runtime. We label these as false positives.

Summary: RVSec is more 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).

5 Qualitative Analysis of Inaccuracy

We discuss causes of inaccuracy (i.e., false positives and false negatives) for RVSec (§5.1), CogniCrypt and CryptoGuard (§5.2), and CryLogger (§5.3). Table 5 summarizes the accuracy of these tools with respect to CWEs.

5.1 Sources of Inaccuracy: RVSec

RVSec achieves high accuracy, as expected for a dynamic analysis. The rare false positives (8 of 556 warnings, Table 3) are due to an overly constrained specification that expects ten thousand or more iterations for Password-Based Encryption, but the SmallCryptoAPIBench labels the usage of one thousand or more iterations as secure (CWE-1391). Overall, ten of RVSec's 22 false negatives (Table 3, column "FN"), are due to lack of test inputs to exercise the crypto API misuses in ApacheCryptoAPIBench. The other twelve false negatives are due to inherent limitations of dynamic analysis. For example, RVSec fails to detect use of string constants to initialize crypto primitives.

5.1.1 Impact of Test Coverage

Low-quality test suites can lead to false negatives with RV, whereas over-approximation can lead to false positives with static analysis. The handcrafted benchmarks (MASCBench, SmallCryptoAPIBench, OWASPBench, and JulietBench) have test suites that execute code with and without crypto API misuses. That is, every crypto API usage is tested, so there is full coverage of code that (mis)uses JCA.

Since the high-coverage test suites in the handcrafted benchmarks likely led to RVSec's high accuracy, we further investigate the impact of lack of test inputs in ApacheCryptoAPIBench's open-source projects. To do so, we measure the 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 JaCoCo. JaCoCo exports coverage measurements for each class, from which we compute IC, BC, and MC. The results are presented in Table 6.

We find that the test suites in ApacheCryptoAPIBench have an average of 44.72% instruction coverage (IC). We do not augment these test suites, so we expected RVSec to find much fewer warnings than CogniCrypt and CryptoGuard. But, RVSec misses only 12 of 51 crypto API misuses in ApacheCryptoAPIBench, despite of the level of coverage. The main reason for RVSec's 0.76 recall on

TABLE 5
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

TABLE 6
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

```
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

ApacheCryptoAPIBench—despite the 44.72% instruction coverage—is that crypto API usage is often confined to few classes that are covered by tests. We find lower RVSec recall in projects where some crypto API usages are not covered by tests. For example, *RVSec missed all 3 API misuses in project ManifoldCF* (Table 2). That project has no tests that cover methods with those crypto API misuses.

> *Impact of Test Coverage on RVSec accuracy:* RVSec achieves 0.76 recall on ApacheCryptoAPIBench despite having only 44.72% instruction coverage, on average, because crypto API usage is often confined to a few covered classes.

5.1.2 Inherent Limitation of RVSec

Seven (of eight) RVSec's false negatives in SmallCryptoAPIBench are due to usage of hard-coded passwords for loading key stores. According to CWE-798, hard-coded passwords can be a threat because *"hard-coded credentials typically create a significant hole that allows an attacker to bypass the authentication that has been configured by the software administrator"* [53]. 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 Figure 10). The recommended best practice is to retrieve passwords from external and protected files or databases [53]. This inherent limitation is one case in which static analyzers like CogniCrypt and CryptoGuard can be used in a complementary manner with RVSec.

> *Main reason for RVSec's false negatives:* It is hard to write RV specifications to check if a variable was initialized to a hard-coded string constant.

5.2 Sources of Inaccuracy: Static Analyzers

Figure 11 shows a crypto API misuse that neither CogniCrypt nor CryptoGuard detect, but which RVSec and CryLogger detect. There, `Cipher` is instantiated by calling the `getAlgorithm` method of class `KeyGenerator`, which returns the string, `"AES"`—`keygen` is instantiated to the output of `KeyGenerator.getInstance("AES")`. But, instantiating `Cipher c` in this way is similar to calling `Cipher.getInstance("AES")`, which specifies just the cipher algorithm, and not its operation mode or padding. The vulnerability occurs because the default mode and padding configuration for `AES` is `ECB/PKCS5Padding`, which may result in disclosing of sensitive information [31].

Creating a `Cipher` as in Figure 11 is insecure, but CogniCrypt and CryptoGuard do not detect this crypto API misuse (CWE-327). But, if one passes `"AES"` string to `Cipher.getInstance()` instead of calling `KeyGenerator.getAlgorithm()`, both tools detect the misuse. To address such false negatives, 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 if multiple method calls are used to initialize a `Cipher` object. Figure 12 shows an example. There, the test case initializes `Cipher c` using the insecure "DES" algorithm (CWE-327). CryptoGuard does not report any issue with the test case in Figure 12. CogniCrypt also misses the first error in this test case (Line 13), but it correctly detects the second one (Line 19). Extending CryptoGuard and CogniCrypt with more advanced inter-procedural data flow analyses may reduce these false negatives in both tools. For instance, FlowDroid is able to detect source-sink flows using different method calls and string manipulations [54].

```java
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

```java
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

CogniCrypt and CryptoGuard report false positives for path-sensitive programs in SmallCryptoAPIBench. Figure 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 Figure 13 as places where the `md` instance of `MessageDigest` *may* be using an insecure implementation (CWE-328). It is questionable whether code similar to the one in Figure 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 cases that we analyze, there is a path in the code that does not satisfy the expected sequence of events that CrySL rules specify for the `Cipher` or `MessageDigest` classes (CWE-325). The code in Figure 14 illustrates the situation, for which CogniCrypt reports warnings like:

```java
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

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 Figure 14 will eliminate this CogniCrypt warning. In this case, the resulting path calls `update()` and `digest()` in `MessageDigest`, 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.

```java
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

CryptoGuard reports 27 false positives and 40 false negatives in OWASPBench. All these mis-classifications 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 Figure 15 and Figure 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 (a challenge for static analysis in general), it wrongly labels the code in Figure 15 as a crypto API misuse (according to the OWASPBench ground truth). For the same reason, CryptoGuard wrongly classifies the code in Figure 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 Figure 15 and Figure 16.

Method calls with string manipulation, path and field sensitivity, and configuration file usage are the main causes of inaccuracy for static analyzers, CogniCrypt and CryptoGuard.

```
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.

5.3 Sources of Inaccuracy: CryLogger

CryLogger generates 53 false positives for OWASPBench. Our manual analysis shows that most of these 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 JulietBench (similar to what we observed with RVSec).

CryLogger's false negatives in JulietBench are due to incomplete usage of a cryptographic primitive (CWE-325). Figure 17 shows an example. There, a call to `digest()` is made without a call to `update()` of `MessageDigest`. CryLogger rules do not address this kind of vulnerability—missing cryptographic step (CWE-325) [24]. CryLogger also does not identify incorrect initialization of seeds from constant byte arrays, and usage of strings to store credentials (CWE-321, CWE-337, CWE-1391). These are the main causes of CryLogger false negatives in SmallCryptoAPIBench.

CryLogger times out on three ApacheCryptoAPIBench projects, leading to 32 false negatives. There are two reasons for these timeouts. First, CryLogger's analysis of Artemis runs out of memory after five minutes, and the operating system kills the process. Second, CryLogger's analysis of Spark and Wicket simply did not finish in 24 hours. So, it seems that CryLogger does not scale well to large projects. Also, printing stack traces during CryLogger analysis produces huge log files. But, printing the stack traces is necessary for us to compare CryLogger with other tools.

6 Discussion

We discuss RVSec overhead on ApacheCryptoAPIBench's test cases (§ 6.1), lessons learned and future work (§ 6.2), and threats to validity (§ 6.3).

```
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.

6.1 RV Overhead

Comparing a dynamic analysis approach like Runtime Verification with static analyses requires discussing RV overhead for crypto API misuse detection. Also, using RV to simultaneously monitor many specifications like we do is more costly than monitoring a specification [17], [55], [56].

Table 7 shows the runtime (in seconds) of tests in ApacheCryptoAPIBench without ("TBase (s)" column) and with ("TRV (s)" column) RV. That table also shows RVSec's overhead ("Overhead (%)" column). RVSec's overhead on these projects ranges from 8.64% (ManifoldCF) to 56.86% (Wicket), with an average of 25.90% and median of 18.32%.

TABLE 7
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

These RVSec overheads may be acceptable, but they will likely grow if more tests are added to improve coverage. Also, we only measure RVSec overhead on one version of 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 versions of a project, e.g., during continuous integration or regression testing [11], [12]. So, using evolution-aware RV to detect crypto API misuses as software evolves could have even lower runtime overheads.

6.2 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 could be complemented with static analysis to check 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

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

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

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

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

to reduce false positives in the presence of configuration files, e.g., by using the recently proposed configuration testing framework, CTests [57].

A recommendation for better usage of RV in crypto API misuse detection. During our research, we identified a design recommendations for dynamic analyses that detect crypto API misuses: they should also instrument crypto API clients, instead of only instrumenting the API as CryLogger does [9]. Instrumenting client code

(a) allows dynamic analysis to report relevant information about crypto API misuses (such as location of the misuse). Such information is essential to help developers understand and fix misuses. We partially fix this CryLogger limitation by changing some of its components to enrich the log files with stack trace information, but doing so increases the log files' size.
(b) 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 specified classes (e.g., Java standard library classes). Our experience in using CryLogger shows that it does not allow such flexibility. Our CryLogger extension does not provide this second benefit, but we manually filter out CryLogger warnings from Java standard library classes.

Issues with Existing Benchmarks. Several test cases in SmallCryptoAPIBench (curated for evaluating static analyses) trigger runtime exceptions. We fix such tests to make them useful for dynamic analyses. For example, in ten tests, we increase the size of byte arrays that are used to configure the initialization vectors of ciphers (Figure 18). This fix is necessary because 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 shown in Figure 19). Several ApacheCryptoAPIBench classes needed fix, though, fortunately, these classes have no use of the JCA library. Our fixes are necessary to build the projects and run their tests. We share all these fixes in our replication package. Setting up OWASPBench and JulietBench went smoother for us than the other benchmarks.

6.3 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. 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 [13], [14], [15].

Our choice of benchmarks may pose an additional threat to validity. However, the five benchmarks in this paper 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 study 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 study 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 warnings. Doing so may add threats to validity, but it improves the benchmark and allows for fairer comparison of RVSec, CogniCrypt, CryptoGuard, and CryLogger. We contacted the authors of ApacheCryptoAPIBench, and they agree on the most critical changes (e.g., our recommendation to treat 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 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. Our new implementation is necessary because CryLogger only reports which crypto API rules a system execution violates. Figure 20 shows a snippet of the outcome of the original CryLogger for the MASCBench. It is not feasible to compute accuracy metrics using the information present in Figure 20. So, our new CryLogger implementation records all crypto API violations together with the program stack trace when violations happen (see Figure 21), allowing us to track client code that misuse crypto APIs.

A downside of our new CryLogger implementation is that it generates large log files, which may have led to timeouts in three ApacheCryptoAPIBench projects. Note that the original version of CryLogger also times out on Spark. So, even the original CryLogger version has scalability issues. We could not obtain a replication package for prior CryLogger evaluation.

```
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

7 Related Work

We discuss work that is most related to ours, including research on crypto API misuses and on combining static and dynamic analysis to identify security vulnerabilities.

7.1 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 [22] and CryptoGuard [7] are two prominent examples of such static analyzers in the literature. CogniCrypt uses rules written in a domain-specific language 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 allows 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. We find that RVSec produces fewer false positives and false negatives than static analyses. So, RVSec can complement static analyses during software development.

7.2 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 [9] 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 for us to change CryLogger's implementation for use in our study.

7.3 Use of Static and Dynamic Analysis to Detect Other Types of Vulnerabilities

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 are program states and the edges are 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 paper also

introduces a framework and tools to support the proposed approach and allow the unified specification, verification, and monitoring of software systems. Their research aims to monitor systems in production, but our goal with RVSec is to identify crypto API misuses during testing. Future work can explore techniques to fix crypto API misuses at runtime.

Handrick et al. [81] present an in-depth analysis on the combination of static and dynamic analysis to detect Android malware—using the *Android Mining Sandbox Approach* [16], [82]. The authors provide evidence that static and dynamic analysis complement each other, improving the overall accuracy of Android malware detection. Our work differs from Handrick et al.'s: we target crypto API misuses, a specific and pressing source of software vulnerability. But, the related work presented in this section may hint at possible directions for integrating static and dynamic analysis tools for detecting crypto API misuses.

8 Conclusions and Future Work

We evaluate the use of RV for detecting crypto API misuses. To do so, we implement RVSec after translating CrySL rules [6], [22] into JavaMOP specifications [10]. Then we use RVSec to identify misuses of the JCA API. We compare RVSec's accuracy with those of state-of-the-art tools CogniCrypt [6], [22], [83], CryptoGuard [7], and CryLogger [9] 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. We also fix CogniCrypt, improving its precision, and 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 *ICSE*, 2016, p. 935–946.

[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, Tech. Rep. BSI TR-02102-1, 2022.

[4] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic APIs," in *S&P*, 2017, pp. 154–171.

[5] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack Overflow considered harmful? the impact of copy&paste on Android application security," in *S&P*, 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 *ECOOP*, 2018, pp. 1–27.

[7] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects," in *CCS*, 2019, p. 2455–2472.

[8] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *CCS*, 2013, p. 73–84.

[9] L. Piccolboni, G. D. Guglielmo, L. P. Carloni, and S. Sethumadhavan, "CryLogger: Detecting crypto misuses dynamically," in *S&P*, 2021, pp. 1972–1989.

[10] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," in *ICSE*, 2012, pp. 1427–1430.

[11] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Roşu, and D. Marinov, "Techniques for evolution-aware runtime verification," in *ICST*, 2019, pp. 300–311.

[12] O. Legunsen, D. Marinov, and G. Roşu, "Evolution-aware monitoring-oriented programming," in *ICSE-NIER*, 2015, pp. 615–618.

[13] 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 *ASE*, 2016, pp. 602–613.

[14] O. Legunsen, N. A. Awar, X. Xu, W. U. Hassan, G. Roşu, and D. Marinov, "How effective are existing Java API specifications for finding bugs during runtime verification?" *ASEJ*, vol. 26, no. 4, pp. 795–837, 2019.

[15] B. Miranda, I. Lima, O. Legunsen, and M. d'Amorim, "Prioritizing runtime verification violations," in *ICST*, 2020, pp. 297–308.

[16] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *ICSE*, 2016, pp. 37–48.

[17] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Şerbănuţă, and G. Roşu, "RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties," in *RV*, 2014, pp. 285–300.

[18] C. Lee, D. Jin, P. O. Meredith, and G. Roşu, "Towards categorizing and formalizing the JDK API," Computer Science Dept., UIUC, Tech. Rep., 2012.

[19] L. Teixeira, B. Miranda, H. Rebêlo, and M. d'Amorim, "Demystifying the challenges of formally specifying API properties for runtime verification," in *ICST*, 2021, pp. 82–93.

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

[21] M. Gabel and Z. Su, "Testing mined specifications," in *FSE*, 2012, pp. 1–11.

[22] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An extensible approach to validating the correct usage of cryptographic APIs," *TSE*, vol. 47, no. 11, pp. 2382–2400, 2021.

[23] S. Afrose, Y. Xiao, S. Rahaman, B. Miller, and D. D. Yao, "Evaluation of static vulnerability detection tools with Java cryptographic API benchmarks," *TSE*, pp. 485–497, 2022.

[24] "Missing cryptographic step," Available from MITRE, CWE-ID CWE-325. [Online]. Available: https://cwe.mitre.org/data/definitions/325.html

[25] 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 *S&P*, 2022, pp. 614–631.

[26] OWASP, "Owasp benchmark," 2022. [Online]. Available: https://owasp.org/www-project-benchmark

[27] NSA, "Juliet benchmark," 2012. [Online]. Available: http://samate.nist.gov/SRD/testsuite.php

[28] A. Yorihiro, P. Jiang, V. Marqués, B. Carleton, and O. Legunsen, "eMOP: A Maven plugin for evolution-aware runtime verification," in *RV*, 2023, pp. to–appear.

[29] D. Hook, *Beginning Cryptography with Java*, 1st ed., GBR, 2005.

[30] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering: Design Principles and Practical Applications*, 2010.

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

[32] S. E. Java Platform, "Java Cryptography Architecture (JCA) Reference Guide," 2022. [Online]. Available: https://docs.oracle.com/en/java/javase/11/security/index.html

[33] S. Krüger, "CogniCrypt – The Secure Integration of Cryptographic Software," Ph.D. dissertation, Universität Paderborn, 2020. [Online]. Available: https://www.bodden.de/pubs/phdKrueger.pdf

[34] "JavaMOP," https://github.com/runtimeverification/javamop.

[35] F. Chen and G. Roşu, "Parametric trace slicing and monitoring," in *TACAS*, 2009, pp. 246–261.

[36] F. Chen, P. O. Meredith, D. Jin, and G. Roşu, "Efficient formalism-independent monitoring of parametric properties," in *ASE*, 2009, pp. 383–394.

[37] "Apache qpid brokerj," 2022. [Online]. Available: https://qpid.apache.org/components/broker-j/index.html

[38] NSA, "Juliet test suite for Java (user guide)," Center for Assured Software, National Security Agency, Tech. Rep., 2012.

[39] S. Afrose, S. Rahaman, and D. Yao, "CryptoAPI-Bench: A comprehensive benchmark on Java cryptographic API misuses," in *SecDev*, 2019, pp. 49–61.

[40] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. D. Yao, and N. Meng, "Example-based vulnerability detection and repair in Java code," in *ICPC*, 2022, p. 190–201.

[41] P. Ferrara, E. Burato, and F. Spoto, "Security analysis of the OWASP benchmark with julia," in *ITASEC*, 2017, pp. 242–247.

[42] B. Mburano and W. Si, "Evaluation of web vulnerability scanners based on OWASP benchmark," in *ICSEng*, 2018, pp. 1–6.

[43] 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?" *TSE*, pp. 288–303, 2022.

[44] D. Beyer, "Software Verification: 10th Comparative Evaluation (SV-COMP 2021)," in *TACAS*, 2021, pp. 401–422.

[45] J. Herter, D. Kästner, C. Mallon, and R. Wilhelm, "Benchmarking static code analyzers," *Reliability Engineering & System Safety*, vol. 188, pp. 336–346, 2019.

[46] "JaCoCo Code Coverage," 2022. [Online]. Available: https://www.eclemma.org/jacoco/

[47] "CrySL repository," 2022. [Online]. Available: https://github.com/CROSSINGTUD/Crypto-API-Rules/

[48] M. Hazhirpasand, M. Ghafari, and O. Nierstrasz, "Java cryptography uses in the wild," in *ESEM*, 2020, pp. 1–6.

[49] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST*, 2016, pp. 80–90.

[50] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *ICSE*, 2018, pp. 433–444.

[51] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! Detecting flaky tests caused by concurrency with Shaker," in *ICSME*, 2020, pp. 301–311.

[52] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *MSR*, 2020, p. 492–502.

[53] "Use of hard-coded credentials," Available from MITRE, CWE-ID CWE-798. [Online]. Available: https://cwe.mitre.org/data/definitions/798.html

[54] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *PLDI*, 2014, pp. 259–269.

[55] D. Jin, P. O. Meredith, and G. Roşu, "Scalable parametric runtime monitoring," UIUC, Tech. Rep., 2012.

[56] P. Meredith and G. Roşu, "Efficient parametric runtime verification with deterministic string rewriting," in *ASE*, 2013, pp. 70–80.

[57] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing configuration changes in context to prevent production failures," in *OSDI*, 2020, pp. 735–751.

[58] "ACM publications policy on research involving human participants and subjects," 2002. [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 *OSS*, 2019, pp. 12–23.

[60] L. Sampaio and A. Garcia, "Exploring context-sensitive data flow analysis for early vulnerability detection," *JSS*, vol. 113, pp. 337–361, 2016.

[61] P. Arteau, "Findsecbugs," Available online. [Online]. Available: https://find-sec-bugs.github.io/

[62] RigsIT, "Xanitizer," Available online. [Online]. Available: https://www.rigs-it.net

[63] SonarSource, "Sonarqube," Available online. [Online]. Available: https://www.sonarqube.org/

[64] NCCGroup, "Visualcodegrepper," Available online. [Online]. Available: https://github.com/nccgroup/VCG

[65] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "SMV-HUNTER: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps," in *NDSS*, 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 *FPS*, 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 *NSS*, 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 *CCS*, 2018, pp. 412–425.

[69] Y. Falcone, S. Currea, and M. Jaber, "Runtime verification and enforcement for Android applications with RV-Droid," in *RV*, 2012, pp. 88–95.

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

[71] P. Daian, Y. Falcone, P. Meredith, T. F. Şerbănuţă, S. Shiriashi, A. Iwai, and G. Roşu, "RV-Android: Efficient parametric Android runtime verification, a brief tutorial," in *RV*, 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 Systems Journal*, vol. 12, no. 2, pp. 1667–1678, 2016.

[73] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez, "BraceAssertion: Runtime verification of cyber-physical systems," in *MASS*, 2015, pp. 298–306.

[74] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Roşu, "ROSRV: Runtime verification for robots," in *RV*, 2014, pp. 247–254.

[75] D. B. d. Oliveira, T. Cucinotta, and R. S. d. Oliveira, "Efficient formal verification for the Linux kernel," in *SEFM*, 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 *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 *DSN*, 2016, pp. 467–478.

[78] X. Shu, D. Yao, N. Ramakrishnan, and T. Jaeger, "Long-span program behavior modeling and attack detection," *TOPS*, 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," *TDSC*, vol. 18, no. 2, pp. 825–842, 2019.

[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," *FMSD*, vol. 51, pp. 200–265, 2017.

[81] F. H. da Costa, I. Medeiros, T. Menezes, J. V. da Silva, I. L. da Silva, R. Bonifácio, K. Narasimhan, and M. Ribeiro, "Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for Android malware identification," *JSS*, vol. 183, p. 111092, 2022.

[82] L. Bao, T. B. Le, and D. Lo, "Mining sandboxes: Are we there yet?" in *SANER*, 2018, pp. 445–455.

[83] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "CogniCrypt: supporting developers in using cryptography," in *ASE*, 2017, pp. 931–936.