RV4ANDROID: An End-to-End Pipeline for Runtime Verification of Cryptographic API Specifications on Contemporary Android Applications

Pedro Henrique Teixeira Costa[1], **Rodrigo Bonifácio**[2]

[1]Computer Science Department – University of Brasília[1]
Brasília – Brazil
phtcosta@gmail.com

[2]Informatics Center – Federal University of Pernambuco
Recife – Brazil[2]
rbonifacio@cin.ufpe.br

Abstract. *The Java Cryptography Architecture (JCA), the standard cryptographic API for Java and Android, is notoriously difficult to use correctly, making cryptographic API misuse a major source of software vulnerabilities. Runtime verification (RV) can detect such misuses during execution, providing concrete evidence of reachable vulnerabilities, but its effectiveness depends on exercising the monitored code. Existing Android RV tooling is also outdated and unable to instrument many contemporary applications. Here we present* RV4ANDROID*, an end-to-end runtime verification framework for contemporary Android applications.* RV4ANDROID *modernizes the instrumentation pipeline, compiles JCA specifications into runtime monitors, injects them into APKs, and automates execution with Android test-generation tools while measuring execution coverage. We evaluated* RV4ANDROID *on 33 official* `gov.br` *apps using an Android test case generation tool (APE) under a one-hour execution budget. We found that 90.9% of the apps (30 of 33) triggered at least one JCA violation at runtime, with weak cryptographic hashing (SHA-1 or MD5) as the strongest security signal. Moreover, 96% of all distinct violations were detected within the first ten minutes of execution, indicating that* RV4ANDROID *can uncover most runtime cryptographic misuses quickly while remaining applicable to contemporary Android applications.*

1. Introduction

Applications entrust their security-critical operations to cryptographic APIs—in the Java ecosystem, chiefly to the Java Cryptography Architecture (JCA). Using these APIs correctly, however, is far from trivial. The APIs impose low-level constraints (e.g., which algorithm and mode to select, in which order to call which methods), and developers often struggle to satisfy them, producing insecure code as a result [Nadi et al. 2016, Krüger et al. 2017]. Consequently, cryptographic API misuses are among the main sources of software vulnerabilities in Java and Android applications. A large-scale static analysis has shown that such misuses are widespread in Android apps and within the Maven ecosystem [Krüger et al. 2021]. Moreover, a misuse seldom announces itself: the application continues to function while, for instance, encrypting data with an insecure mode or using an improperly initialized cipher, allowing the problem to remain unnoticed during functional testing.

To mitigate this threat, previous work has proposed both static and dynamic analysis techniques for detecting cryptographic API misuses. Static analysis tools such as CogniCrypt and Cryptoguard [Krüger et al. 2017, Krüger et al. 2021, Rahaman et al. 2019] check programs against a set of rules that describe either the correct or incorrect usage of cryptographic APIs. Since static analysis must approximate program behavior to remain computationally tractable, it may report infeasible execution paths and consequently produce false positives, flagging misuses that cannot actually occur at runtime. Runtime verification (RV) approaches the same problem from a complementary perspective. Monitors synthesized from formal specifications using, for instance, Monitoring Object Programming (MOP) and tools such as JavaMOP [Jin et al. 2012] and RV-Monitor [Luo et al. 2014] observe concrete executions and report violations as they occur. Unlike static analysis, RV reports only misuses exercised during execution, thereby providing concrete evidence that a reported misuse is actually reachable. This stronger evidence comes at the cost of depending on execution coverage: code that is not executed cannot be verified.

Applying runtime verification to Android applications presents several practical challenges. First, comprehensive developer-written test suites are often unavailable [Pecorelli et al. 2022], making it necessary to rely on automated test-generation tools, such as Monkey [Patel et al. 2018], DroidBot [Li et al. 2017], APE [Gu et al. 2019], and similar approaches, to achieve sufficient execution coverage. Second, Android applications are distributed as APKs rather than source code, requiring monitors to be woven directly into DEX bytecode.[1] Third, existing runtime verification tooling is outdated. RV-Android, which introduced parametric runtime verification for Android as a userspace monitoring library built on RV-Monitor [Daian et al. 2015], demonstrated the feasibility of runtime verification on the platform. However, the Android ecosystem has evolved substantially since its introduction, while RV-Android has remained largely unchanged for more than a decade. Contemporary Android applications rely on significantly different build, packaging, signing, and bytecode-processing pipelines, preventing RV-Android from instrumenting many modern applications without substantial modifications. Moreover, RV-Android provides no built-in integration with automated test-generation tools, requiring users to manually coordinate instrumentation and test execution to exercise the monitored application. We present additional background information in Section 2.

In this paper, we present RV4ANDROID, which addresses the aforementioned challenges. It modernizes the instrumentation pipeline for contemporary Android applications while automating the entire runtime verification workflow, from monitor synthesis and APK instrumentation to emulator deployment, automated execution with Android test-generation tools, coverage measurement, and violation reporting (Section 3 presents RV4ANDROID). We evaluated RV4ANDROID on 33 official `gov.br` apps, executing each with the APE test generation tool under a one-hour execution budget. The evaluation showed that 30 apps (90.9%) triggered at least one JCA MOP violation, that weak hashing (SHA-1 or MD5) was the most recurring misuse, and that most distinct violations were detected within the first ten minutes of execution (Section 4).

[1]Dalvik (DEX) bytecode format: `https://source.android.com/docs/core/runtime/dalvik-bytecode`.

Altogether, this paper makes two main contributions: (a) RV4ANDROID, an end-to-end runtime verification pipeline for contemporary Android applications; and (b) an empirical study demonstrating its effectiveness by characterizing the prevalence, nature, and time to detection of JCA misuses in official `gov.br` apps.

2. Background

Modern software systems rely heavily on cryptographic APIs to implement security mechanisms such as encryption, hashing, digital signatures, and secure communication. Correct use of these APIs, however, is notoriously difficult [Nadi et al. 2016]. Developers must not only invoke methods in the proper order, but also select secure algorithms, satisfy parameter constraints, and avoid deprecated or insecure configurations. Empirical studies have shown that cryptographic API misuses are widespread and may introduce security vulnerabilities [Nadi et al. 2016, Amann et al. 2019, Krüger et al. 2021]. To detect such misuses, researchers have developed specification languages that capture correct API usage patterns, together with analysis tools that use these specifications to distinguish valid from invalid usage.

These specification-based tools typically rely on either static or dynamic analysis. Static analyzers inspect source code or bytecode without executing the program, providing broad coverage but often reporting warnings along infeasible execution paths. Runtime verification is one form of dynamic analysis that checks API usage over concrete executions by monitoring the events that occur during program execution [Leucker and Schallhart 2009]. Although runtime verification cannot reason about unexecuted paths, it naturally avoids false positives caused by infeasible paths and can verify properties that depend on runtime values or dynamic program behavior.

Runtime verification is commonly implemented using Monitoring-Oriented Programming (MOP) [Chen and Rosu 2007]. In MOP, developers specify the execution events to observe, the properties relating those events—for example, a finite-state machine describing valid method-call sequences—and the actions to perform when a property is satisfied or violated. JavaMOP [Jin et al. 2012] implements this paradigm for Java. From a specification, JavaMOP generates AspectJ [Kiczales et al. 2001] instrumentation that intercepts the specified API calls and translates them into monitoring events, while RV-Monitor [Luo et al. 2014] generates efficient runtime monitors that process these events during program execution. Monitoring is parametric: independent monitor instances can be maintained for distinct objects, such as individual `Cipher` or `MessageDigest` instances.

The specifications used by RV4ANDROID are derived from CrySL rules [Krüger et al. 2021], a specification language designed for cryptographic APIs. CrySL enables security experts to define method-ordering constraints, parameter restrictions, and other usage requirements for each API. The CrySL rule set developed for the Java Cryptography Architecture (JCA) is widely used by static analyzers such as CogniCrypt [Krüger et al. 2017, Krüger et al. 2021]. RV4ANDROID consumes an equivalent set of specifications translated into JavaMOP [Torres et al. 2023], allowing cryptographic usage constraints traditionally checked through static analysis to be verified over concrete executions of Android applications. These JavaMOP specifications were previously validated on Java programs and benchmarks [Torres et al. 2023].

3. RV4ANDROID: Architecture

RV4ANDROID, implemented in Java and Python, takes two inputs: formal specifications of correct API usage—for the JCA, the ones translated from CrySL—and one or more Android application packages (APKs). The framework supports two execution modes. In audit mode, it analyzes a single APK by instrumenting it and executing it with a selected test-generation tool for a user-defined time budget, reporting the detected API-usage violations. In batch mode, RV4ANDROID executes the same pipeline over a collection of APKs, optionally varying the test-generation tool and execution budget across experimental units. Besides consolidating the detected violations, this mode also exports coverage information and other execution statistics, making it suitable for large-scale empirical studies. Regardless of the execution mode, a run proceeds in two stages: preprocessing, which synthesizes the executable monitors and instruments the APKs, and execution, which runs the instrumented applications and consolidates the observations produced by the monitors. Figure 1 sketches this workflow and the modules that implement it; the remainder of this section describes these modules in order.

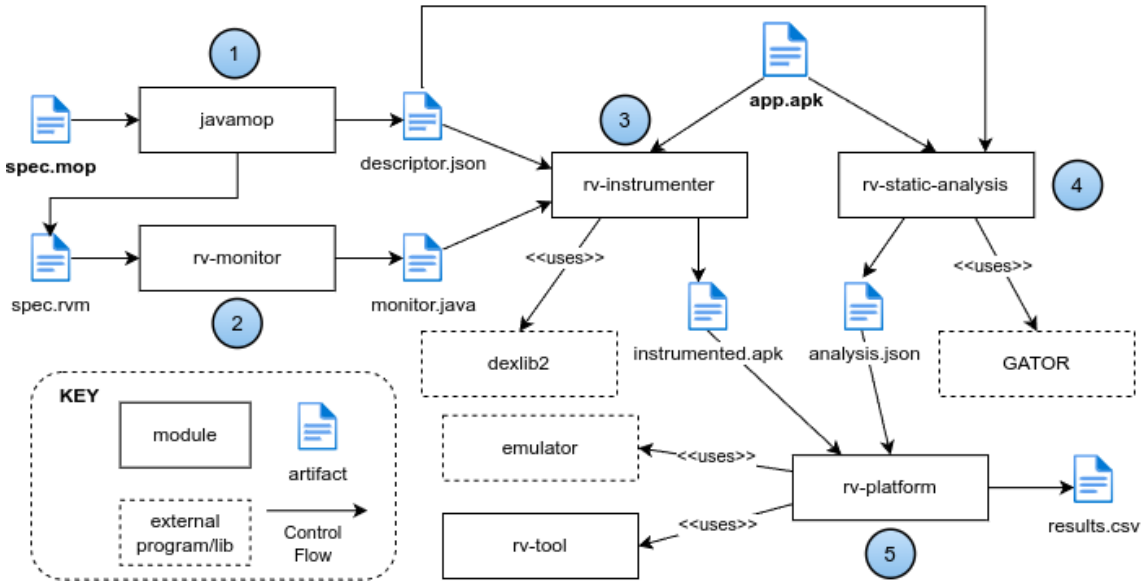

Figure 1. Overview of the RV4ANDROID pipeline. A specification and an application package (`app.apk`) enter a preprocessing stage—JavaMOP and RV-Monitor synthesize the monitor, `rv-instrumenter` weaves it into the APK, and `rv-static-analysis` computes the reachable methods—followed by an execution stage in which `rv-platform` runs the instrumented app on an emulator. The single output node is schematic: the execution stage emits several result files (Section 3.2), of which `errors.csv` records the detected violations.

3.1. Preprocessing: from specifications to an instrumented app

The preprocessing stage builds, from each specification, a monitor able to observe the application at run time, and embeds that monitor into the APK.

Monitor synthesis falls to JavaMOP and RV-Monitor. JavaMOP is incorporated as a vendored fork—the upstream project copied into the tool's own source tree—patched locally so that its output also includes a JSON descriptor of the events to be monitored (the

`descriptor.json` of Figure 1). That descriptor is what the instrumenter consumes: with it, the module identifies the method calls to intercept without parsing AspectJ pointcuts. RV-Monitor, incorporated in the same way, synthesizes the Java monitor classes from the `.rvm` specification. Together, the two modules produce the monitor and the descriptor that `rv-instrumenter` weaves into the application.

The `rv-instrumenter` module rewrites the APK so that the monitored calls report their events to the monitor at run time. It edits the application's DEX bytecode in place, using the dexlib2 library.[2] Note that rewriting the DEX directly avoids the round trip: disassembling the APK, weaving the monitor over an intermediate Java-bytecode form, and recompiling back to DEX. That round trip fails on APKs optimized by R8, because some idioms that R8 emits cannot be expressed in intermediate bytecode that satisfies the type-consistency rules of the JVM specification. For this reason, the verifier of the Android runtime then rejects the recompiled application at load time.

After instrumentation, the `rv-static-analysis` module computes a static over-approximation of the application's behavior; it runs only for the APKs whose instrumentation succeeded, and it analyzes the original APK, not the instrumented one. The module builds on GATOR [Yang et al. 2015], a static-analysis framework for Android, through a fork that integrates a client component (`RvsecAnalysisClient`) and runs on Soot 4.7.1. Its starting point is the full method list of the application's own package (identified automatically, with the manifest as fallback), excluding the generated `R`, `R$*`, and `BuildConfig` classes.

This list is the baseline for code coverage: its size is the denominator of the fraction of the application's code exercised at run time. A reachability analysis annotates the list, marking, for each method, whether it is reachable from the application's entry points and whether it reaches a monitored operation. The entry points are the lifecycle handlers and the public and protected methods of the components declared in the manifest (activities, services, broadcast receivers, and content providers). The module writes the annotated list as JSON (the `analysis.json` of Figure 1).

3.2. Execution: running the instrumented apps

The `rv-platform` module orchestrates the execution stage in both audit and batch modes. It takes three inputs: the run configuration, the instrumented APKs produced during preprocessing, and the static-analysis JSON generated by `rv-static-analysis`. In audit mode, `rv-platform` executes a single instrumented APK with a selected test-generation tool. In batch mode, it schedules multiple execution tasks across collections of APKs, test-generation tools, execution budgets, and repetitions, enabling controlled empirical evaluations. Each task consists of executing one instrumented APK with one test-generation tool on an emulator for a fixed execution budget (a timeout). We have successfully integrated different test generation tools into the RV4ANDROID pipeline, including APE [Gu et al. 2019], DroidBot [Li et al. 2017], and Monkey [Patel et al. 2018].

From each execution task, `rv-platform` collects two kinds of information. Coverage is computed by combining the events logged under the `RVSEC-COV` logcat

[2]dexlib2 is the Dalvik-bytecode manipulation library of the `smali`/`baksmali` project: `https://github.com/google/smali`.

tag with the annotated method list produced during static analysis. The subset of methods that directly reach a monitored operation defines the denominator of the monitored-coverage metric. Violations—that is, API-usage misuses detected by the runtime monitor rather than application crashes—are recovered from the `RVSEC` logcat tag and recorded in `errors.csv`. Finally, `rv-platform` consolidates the collected information into a set of result files. The single output node in Figure 1 represents this collection of outputs, among which `errors.csv` is the primary artifact used in the empirical analyses reported in this paper.

3.3. Command Line Interface

We **demonstrate** RV4ANDROID by walking through a single run on one example application, from the command line to the result files. The orchestration is handled by the `rv-experiment` module, which ties the preprocessing and execution stages together behind one command-line interface. This execution requires an Android emulator and the Android SDK; two environment variables, `RVSEC_HOME` and `ANDROID_HOME`, point the tool at its own resources and at the SDK.

The example is `cryptoapp`, a small Android app that exercises the JCA; we analyze it against the JCA specification set. The `run` subcommand launches the analysis, naming the test-generation tool (here, Monkey), the specification set, and the directory holding the APKs:

```
uv run rv-experiment run --tools monkey \
    --specification-set jca --apks-dir ./apks_examples
```

That one command covers both stages: the tool instruments the APK, drives it on an emulator with Monkey, and logs what the monitors observe.

When the run finishes, the evaluator opens two of the result files. `errors.csv` lists the JCA misuses the monitors detected during the run, one row per violation, naming the violated specification and the class and method where it occurred. `coverage.csv` reports how much of the reachable-methods baseline the run exercised. The RV4ANDROID assessment of Section 4 rests on runs of exactly this kind, repeated over the `gov.br` apps, each driven by APE.

4. Empirical Assessment

We structure the RV4ANDROID evaluation around three research questions:

- **RQ1 (prevalence).** Does RV4ANDROID detect JCA misuse across a corpus of official government apps under a single automated-exploration configuration?
- **RQ2 (nature and defensibility of the detected misuses).** What types of violation does it detect, and which represent a defensible signal rather than known specification-modeling noise?
- **RQ3 (time to detection).** How much of the execution budget does RV4ANDROID, driven by APE, actually need to expose the distinct violations it detects?

We answer these questions by applying RV4ANDROID to 33 official `gov.br` apps, executing each JCA-instrumented APK with the APE test generation tool under a one-hour execution budget, using the Android API level 30 and the `x86_64`

emulator,. We selected APE because a preliminary evaluation showed that it consistently achieved the highest JCA-specific coverage among eleven test-generation tools across multiple execution budgets. Although a comprehensive evaluation across multiple test-generation tools and application corpora is beyond the scope of this paper, official Brazilian government apps provide a representative deployment scenario: they process citizens' personal data under Brazil's General Data Protection Law (LGPD) [Brasil, Presidência da República 2018].

Out of the 33 apps, 30 (90.9%) triggered at least one JCA MOP violation, for 424 distinct violations (RQ1). Methods executed per app ranged from 176 to a median of 4,529. Apps with no reported violations tend to lie at the low end of that range, suggesting that the absence of findings should not be interpreted as evidence of correct JCA usage. Overall, these results indicate that JCA misuse is prevalent among the analyzed government applications. Table 1 breaks the violations down by JCA specification rather than by violation type (RQ2), from `MessageDigestSpec` (224 distinct violations) to `MacSpec` (1).

Table 1. JCA MOP violations detected across the 33 `gov.br` apps, by JCA specification: apps affected, distinct violations, and raw occurrences.

spec	apps affected	distinct	raw occurrences
MessageDigestSpec	28	224	37,260
KeyStoreSpec	9	53	13,566
SSLContextSpec	17	51	10,671
TrustManagerFactorySpec	17	34	6,972
SecureRandomSpec	19	25	5,823
CipherSpec	6	36	5,581
MacSpec	1	1	12

Weak hashing is the strongest finding in Table 1: under `MessageDigestSpec`, `UnsafeAlgorithm` violations (weak hashes such as SHA-1 or MD5) affect 28 of the 33 apps, corresponding to 96 distinct violations. An in-depth analysis of some warnings also reveals that existing JCA specifications require Android-specific adaptations rather than indicating security flaws in the analyzed applications. For example, `SSLContextSpec` flags Android's recommended use of `TLS` instead of explicit `TLSv1.2` or `TLSv1.3` literals, while `KeyStoreSpec` flags the Android-specific `AndroidKeyStore` implementation. Overall, the evaluation distinguishes actionable security findings from specification mismatches, highlighting opportunities to adapt JCA specifications to Android.

RQ3 asks how much of the one-hour budget RV4ANDROID actually needs. For each of the 424 distinct violations across the 30 apps, we measure the elapsed execution time to first detection. The median is 1s, and 63.9% of the distinct violations are already detected within the first second of execution. Figure 2 plots the cumulative share of distinct violations detected against elapsed time: the curve rises steeply at the start and then flattens, with 96.0% of the 424 violations detected within the first ten minutes. The last distinct violation is not detected until 3,274s (55 minutes) into its app's execution, still within the one-hour budget. Thus, while most violations are exposed quickly, the full

execution budget is required to uncover the set of distinct violations found.

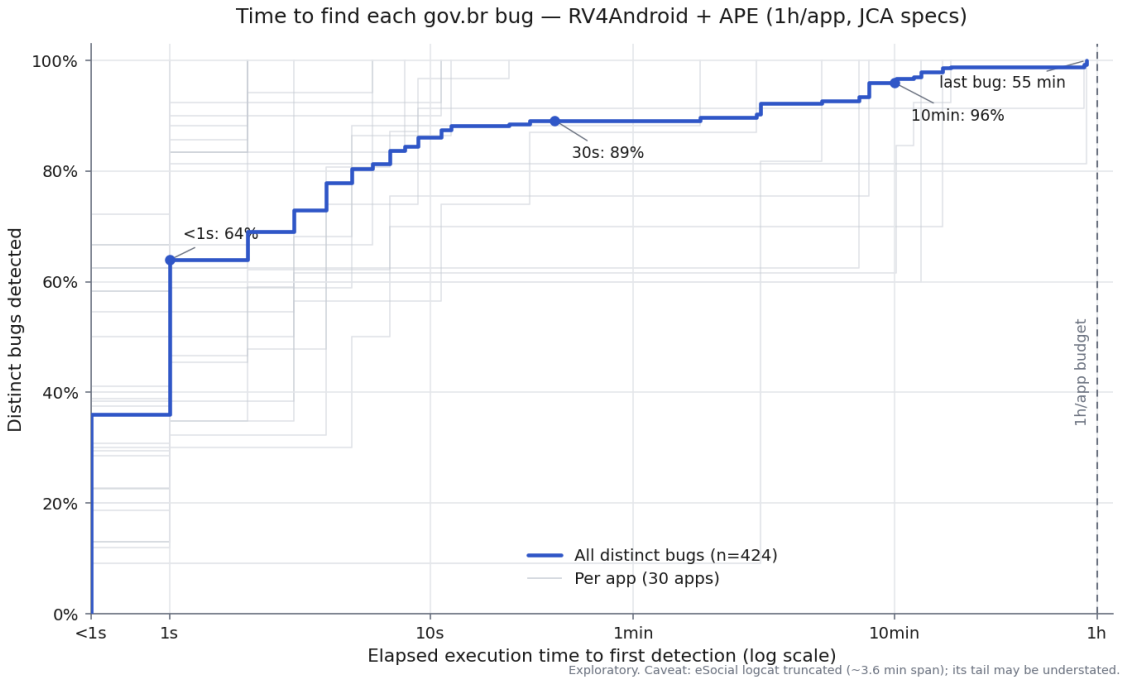

Figure 2. Cumulative share of the 424 distinct violations detected by RV4ANDROID against elapsed execution time (log scale). The bold curve aggregates across the 30 apps with at least one violation; the faint curves trace each app on its own. The dashed line marks the one-hour execution budget.

5. Final Remarks

This paper presented RV4ANDROID, an end-to-end framework for runtime verification of contemporary Android applications. By modernizing the instrumentation pipeline and automating the complete runtime-verification workflow, RV4ANDROID enables API-usage specifications to be checked directly on Android APKs using automated test-generation tools. We evaluated RV4ANDROID on 33 official `gov.br` apps using JCA specifications. 30 applications (90.9%) triggered at least one runtime violation, with weak cryptographic hashing emerging as the strongest security signal. Moreover, most distinct violations were exposed within the first ten minutes of execution, suggesting that runtime verification can provide actionable security evidence with modest execution budgets. Although our empirical assessment focused on the Java Cryptography Architecture, the specification set is an input to the framework rather than a built-in assumption. Consequently, RV4ANDROID provides a general infrastructure for bringing specification-based runtime verification to contemporary Android applications and can be readily extended to other API-usage domains.

Data and Artifact Availability

Artifacts accompanying this paper are publicly available.

- demonstration video (YouTube): `https://shorturl.at/5G2yW`
- experimental package (GitHub): `https://shorturl.at/MGmo2`
- RV4ANDROID implementation (GitHub): `https://shorturl.at/j3AZd`

References

Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N., and Mezini, M. (2019). A systematic evaluation of static api-misuse detectors. *IEEE Trans. Software Eng.*, 45(12):1170–1188.

Brasil, Presidência da República (2018). Lei nº 13.709, de 14 de agosto de 2018 (Lei Geral de Proteção de Dados Pessoais – LGPD), art. 46. `https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm`.

Chen, F. and Rosu, G. (2007). Mop: an efficient and generic runtime verification framework. In Gabriel, R. P., Bacon, D. F., Lopes, C. V., and Jr., G. L. S., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 569–588. ACM.

Daian, P., Falcone, Y., Meredith, P. O., Serbanuta, T., Shiraishi, S., Iwai, A., and Rosu, G. (2015). Rv-android: Efficient parametric android runtime verification, a brief tutorial. In Bartocci, E. and Majumdar, R., editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*, pages 342–357. Springer.

Gu, T., Sun, C., Ma, X., Cao, C., Xu, C., Yao, Y., Zhang, Q., Lu, J., and Su, Z. (2019). Practical GUI testing of android applications via model abstraction and refinement. In Atlee, J. M., Bultan, T., and Whittle, J., editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 269–280. IEEE / ACM.

Jin, D., Meredith, P. O., Lee, C., and Rosu, G. (2012). Javamop: Efficient parametric runtime monitoring framework. In Glinz, M., Murphy, G. C., and Pezzè, M., editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1427–1430. IEEE Computer Society.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of aspectj. In Knudsen, J. L., editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer.

Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., Göpfert, F., Günther, F., Weinert, C., Demmler, D., and Kamath, R. (2017). Cognicrypt: supporting developers in using cryptography. In Rosu, G., Penta, M. D., and Nguyen, T. N., editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 931–936. IEEE Computer Society.

Krüger, S., Späth, J., Ali, K., Bodden, E., and Mezini, M. (2021). Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Trans. Software Eng.*, 47(11):2382–2400.

Leucker, M. and Schallhart, C. (2009). A brief account of runtime verification. *J. Log. Algebraic Methods Program.*, 78(5):293–303.

Li, Y., Yang, Z., Guo, Y., and Chen, X. (2017). Droidbot: a lightweight ui-guided test input generator for android. In Uchitel, S., Orso, A., and Robillard, M. P., editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 23–26. IEEE Computer Society.

Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P. O., Serbanuta, T., and Rosu, G. (2014). Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In Bonakdarpour, B. and Smolka, S. A., editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 285–300. Springer.

Nadi, S., Krüger, S., Mezini, M., and Bodden, E. (2016). Jumping through hoops: why do java developers struggle with cryptography apis? In Dillon, L. K., Visser, W., and Williams, L. A., editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 935–946. ACM.

OWASP. MASWE-0021: Improper hashing. `https://mas.owasp.org/MASWE/MASVS-CRYPTO/MASWE-0021/`. OWASP Mobile Application Security Weakness Enumeration (MASWE), category MASVS-CRYPTO-1.

Patel, P., Srinivasan, G., Rahaman, S., and Neamtiu, I. (2018). On the effectiveness of random testing for android: or how I learned to stop worrying and love the monkey. In Bai, X., Li, J. J., and Ulrich, A., editors, *Proceedings of the 13th International Workshop on Automation of Software Test, AST@ICSE 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 34–37. ACM.

Pecorelli, F., Catolino, G., Ferrucci, F., Lucia, A. D., and Palomba, F. (2022). Software testing and android applications: a large-scale empirical study. *Empir. Softw. Eng.*, 27(2):31.

Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., and Yao, D. D. (2019). Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In Cavallaro, L., Kinder, J., Wang, X., and Katz, J., editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2455–2472. ACM.

Torres, A., Costa, P. H. T., Amaral, L. H. V., Pastro, J., Bonifácio, R., d'Amorim, M., Legunsen, O., Bodden, E., and Canedo, E. D. (2023). Runtime verification of crypto apis: An empirical study. *IEEE Trans. Software Eng.*, 49(10):4510–4525.

Yang, S., Zhang, H., Wu, H., Wang, Y., Yan, D., and Rountev, A. (2015). Static window transition graphs for android (T). In Cohen, M. B., Grunske, L., and Whalen, M., editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 658–668. IEEE Computer Society.