2017 IEEE 28th International Symposium on Software Reliability Engineering

# Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study

Alexandre Braga, Ricardo Dahab
Institute of Computing
State University of Campinas, Brazil
ambraga@cpqd.com.br, rdahab@ic.unicamp.br

Nuno Antunes, Nuno Laranjeiro, Marco Vieira
CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
nmsa@dei.uc.pt, cnl@dei.uc.pt, mvieira@dei.uc.pt

***Abstract*—The incorrect use of cryptography is a common source of critical software vulnerabilities. As developers lack knowledge in applied cryptography and support from experts is scarce, this situation is frequently addressed by adopting static code analysis tools to automatically detect cryptography misuse during coding and reviews, even if the effectiveness of such tools is far from being well understood. This paper proposes a method for benchmarking static code analysis tools for the detection of cryptography misuse, and evaluates the method in a case study, with the goal of selecting the most adequate tools for specific development contexts. Our method classifies cryptography misuse in nine categories recognized by developers (weak cryptography, poor key management, bad randomness, etc.) and provides the workload, metrics and procedure needed for a fair assessment and comparison of tools. We found that all evaluated tools together detected only 35% of cryptography misuses in our tests. Furthermore, none of the evaluated tools detected insecure elliptic curves, weak parameters in key agreement, and most insecure configurations for RSA and ECDSA. This suggests cryptography misuse is underestimated by tool builders. Despite that, we show that it is possible to benefit from an adequate tool selection during the development of cryptographic software.**

***Index Terms*—static analysis tools, cryptography, benchmarking, software security**

## I. Introduction

Cryptography misuse is a common source of vulnerabilities introduced in software during development efforts [1], being mostly related to coding activities, but also found in design and architecture. Frequently, developers lack knowledge in applied cryptography, barely having support from experts when solving issues related to it.

Nowadays, many common software applications have strong security needs related to cryptography. For instance, mobile apps for secure end-to-end instant messages [2], authenticated encryption in SMS [3], and encrypted file systems with secure deletion [4], go far beyond traditional use cases for cryptography (e.g., SSL security, file encryption, or password protection) and require cryptographic protocols to be blended into their functionality. On the other hand, in ordinary software development, access to experts in applied cryptography is frequently restricted by tight schedules and limited budgets. Because these experts are rare, expensive, and busy, their involvement in software development is usually delayed to testing before deployment [5].

Static code analysis tools can help by detecting cryptography misuse during coding or reviews [6]. Such tools should be able not only to assist developers in daily coding activities, but also to support experts in efficiently finding potential issues during reviews and architecture analysis, according to structured methodologies [5]. A key point for the success of these methodologies is the correct choice of adequate static code analysis tools for specific development scenarios. For instance, in a simple scenario, ordinary developers, novice in cryptography, may need to code common use cases, requiring a precise, but not so complete, tool. In another scenario, knowledgeable developers may work together with cryptography experts to build sophisticated protocols blended into application functionality, requiring the most precise and complete toolkit available.

**This paper proposes a benchmark for static code analysis tools (SCATs) focusing on their support for detecting cryptography misuse**. Roughly speaking, our approach computes metrics (recall, precision, and f-measure) from measurements of tool execution on a set of test cases for cryptography misuse. The objective is twofold. First, to compare different tools directly, showing their limitations and strengths. Second, and more importantly, to determine how well tools perform in the context of cryptographic software development.

We have evaluated five free SCATs in order to answer the questions of how and to which extent cryptography misuse is caught by free SCATs currently available to developers. Free tools are readily available (no purchase of licences is required) and, in many cases, are the first and only option for ordinary developers. We found that the union of misuses detected by all five tools cover only about 35% of crypto misuses in our test cases. We also found that, in general, tools perform better in simple misuses regarding weak cryptography and bad randomness, and worse in issues for key management and program design flaws. Additionally, we found that the tools tested are unable to detect non-trivial misuses for insecure curve selection in Elliptic Curve Cryptography (ECC), weak parameters for key agreement with Diffie–Hellman (DH) and ECDH, misconfigured digital signatures with ECDSA, and many insecure configurations for RSA.

Our method uses a pondered sum of metrics (weighted metrics) to show how likely SCATs can detect expected cryptography misuse in certain application types. Also, such metrics capture development needs concerning SCATs, helping to assemble adequate toolkits for cryptographic software. We observed that weighted precision and weighted recall (in our method, the weighted versions of recall and precision) are better metrics for expressing tool strengths and limitations than non-weighted metrics. Finally, we recommend SCAT usages for specific scenarios of cryptographic software development, according to tool performance during benchmarking.

To the best of our knowledge, this work is the first practical evaluation of SCATs concerning support for cryptography usage, including a detailed set of test cases for cryptography misuse. The main contributions of this work are the following: (1) a method for SCAT evaluation and comparison in detecting cryptography misuse; (2) a set of realistic test cases for cryptography misuse in Java; (3) an assessment of free SCATs showing actual gaps in crypto misuse coverage; (4) the evaluation of the tools according to our method; and (5) recommendations for SCAT usage in specific development scenarios. Java was chosen for this instance of our method because this programming language has a well-known and stable cryptographic API and was adopted by the Android platform, being extensively used worldwide by ordinary developers.

The outline of this paper is as follows. Section II presents related work and Section III overviews our method. Section IV presents likely cryptography misuse for applications, Section V explains development contexts for cryptography, while Section VI describes benchmarking scenarios for cryptography. Section VII details the assessment results and Section VIII instantiates a benchmark as a case study. Section IX discusses our findings and analyses threats to validity. Section X concludes the paper.

## II. Related work

This section presents related works on cryptography misuse, SCATs for cryptography, and metrics for evaluation of vulnerability detection tools.

### A. Cryptography Misuse

Lazar et al. [7] show that only 17% of cryptography vulnerabilities are inside software libraries, the other 83% are misuse of libraries. Also, Chatzikonstantinou et al. [8] concluded that about 88% of Android apps misuse cryptography. Braga and Dahab [1] investigated the occurrence of cryptography misuse in on-line forums for cryptographic programming, showing that crypto misuses have high probabilities (e.g., 90% for Java).

For Egele et al. [9] and Shuai et al. [10], deterministic encryption is the most common misuse when a block cipher (e.g., AES or 3DES) uses the Electronic Code Book (ECB) mode. Asymmetric deterministic encryption with non-randomized RSA is also a misuse [11]. Hardcoded or repeated Initialization Vectors (IV) and hard-coded seeds for Pseudorandom Number Generators (PRNGs) are also frequent [9]. Other misuses come from exchanging operation modes without considering IV needs [1].

Georgiev et al. [12] and Fahl et al. [13] showed that libraries for SSL/TLS allow programmers to ignore parts of certificate validation, adding vulnerabilities exploitable by man-in-the-middle attacks. Recent studies showed misuses related to weak or misplaced parameters for RSA [14], key agreement misconfiguration (e.g., DH and ECDH) [15], and Elliptic Curve Cryptography (ECC) [16], [17] as well.

Nadi et al. [18] observed that developers usually implement simple use cases (e.g. user authentication, storage of login data, secure connections, and data encryption), but face difficulties when using low-level Java APIs. For instance, Shuai et al. [19] discovered that password protection in Android is greatly affected by cryptography misuse. Finally, Braga and Dahab [1] found that the most widespread misuse is weak cryptography, affecting several crypto use cases, and the most troublesome use case is encrypting data at rest, which is affected by several misuses.

### B. Static Code Analysis for Cryptography

The way cryptography has been approached by Secure Software Development Lifecycles (SSDL) was recently studied [5], [6]. The conclusion is that, in general, development teams are aware of cryptography sensitivity, but: (i) they fail to adopt organized lifecycles intended to develop secure crypto software; and (ii) they lack expert help to assure quality at different stages of the development process. Furthermore, the development of secure crypto software still lacks professional tool support for most issues [6].

Domain-specific tools target the coding and testing of cryptographic algorithms and development of cryptographic libraries [6]. On the other hand, emerging research targets tools for development of cryptography-enabled functionality with established and standardized algorithms, as well as trusted libraries and frameworks, including tools for secure programming [20], [21] and verification [9], [10], [22], with particular interest in testing tools for crypto misuses [13], [23], [24].

Current coding standards [25], [26] do offer simple advice and general rules against cryptography misuse that could be automated by simple tools. Also, there are ordinary SCATs with simple rules for crypto misuses (e.g., [27]–[31]), providing late detection [32]. Highly specific issues (usually not detected by ordinary tools) have been addressed by academic prototypes, including SCATs for Android [9], [10], [13] and iOS [33] apps, Java code [21], and misuse of SSL libraries [22].

The Juliet test suite [34] offers synthetic examples of insecure coding and is part of a bigger set of test cases, provided by the Software Assurance Reference Dataset (SARD) [35]–[37], and used to evaluate SCATs [38]–[41]. However, these test suites are quite limited, being restricted to risky or broken encryption and hash functions (weak cryptography), and to deterministic PRNGs (bad randomness).

The open questions answered in this paper are how and to which extent free SCATs detect cryptography misuse.

### C. Benchmarking Vulnerability Detection Tools

Benchmarking vulnerability detection tools has gained attention in areas where long-term security assurance is more important than detection of fancy vulnerabilities. Considering the evaluation of static code analysis tools in general, existing initiatives (such as the Software Assurance Metrics And Tool Evaluation - SAMATE [35], [42]) tried to establish a minimum set of test cases as baseline for vulnerability coverage. Recent initiatives [43] adopted a metrics-based approach in which general behavior for metrics (e.g., precision and recall) is more important than detection of specific vulnerabilities. Unfortunately, none of these initiatives focus on cryptography misuse.

Vulnerability detection tools classify parts of the target application in one of two classes: vulnerable or non-vulnerable. In this way, evaluation metrics are based on raw measures obtained from calculating a confusion matrix which represents the possible outcomes for each classified instance, according to a reference oracle of test cases. In Table I, True positives (TP) is the number of actual vulnerabilities detected by evaluated tools; False positives (FP) is the number of vulnerabilities detected that, in fact, do not exist (false alarms); False negatives (FN) is the number of true, but undetected vulnerabilities (omissions); and True negatives (TN) is the number of tests without actual vulnerabilities.

Antunes and Vieira [44] studied in depth the effectiveness of various metrics for benchmarking security tools and concluded that benchmarks for vulnerability detection tools should consider metrics that are able to capture the effectiveness of the tools in the concrete scenario where those tools are to be used. According to them [44], three metrics commonly used in benchmarks and easily computed from raw measures are Recall, Precision, and f-measure, but these may not be the best ones in all cases. Antunes and Vieira [45] also addressed the practical problem of benchmarking vulnerability detection tools in web services environments. They defined two benchmarks for vulnerability detection tools targeting the well-known SQL Injection vulnerability. One benchmark uses a predefined set of test cases, and the other is based on a user-defined selection of tests. Their results showed that the benchmarks accurately reflect the effectiveness of vulnerability detection tools. Their study, however, holds only for a single type of vulnerability.

TABLE I
CONFUSION MATRIX FOR BINARY CLASSIFICATION.

| **Oracle** | **Reported by evaluated tool** | |
|---|---|---|
| **Test case** | **Condition Positive (P)** | **Condition Negative (N)** |
| **Positive** | True Positive (TP) | False Negative (FN) |
| **Negative** | False Positive (FP) | True Negative (TN) |

Static code analysis of programs is undecidable in general [46]. In spite of that, many working solutions have been applied in practice, but in quite limited ways, as shown by several works that help understanding the limits of this technology. The following paragraphs analyze those related to ours.

Kupsch and Miller [47] compared manual and automated vulnerability assessment and found that from all the vulnerabilities discovered by manual assessment, tools found only simple implementation bugs, but did not find issues requiring deep understanding of code or design.

Diaz and Bermejo [38] compared tools against the SARD test suites [35] and analyzed results using known metrics (e.g., recall, precision, F-M), finding an average precision of 0.7 and an average recall of 0.527. They concluded [38] that it is not yet possible to standardized the behavior of SCATs, because the differences in design and technologies among current tools lead them to detect different kinds of vulnerabilities.

Mahonar et al. [48] assessed SCATs' ability to detect vulnerabilities in source code, concluding that it is hard to benchmark tools just by looking at the results produced against a certain test suite, confirming that tools find less flaws as the cyclomatic complexity of source code increases.

Goseva-Popstojanova and Perhinschi [39] used the Juliet test suite [34] to evaluate three commercial SCATs. They found [39] that 27% of C/C++ vulnerabilities and 11% of Java vulnerabilities were missed by all tools. They also found an overall recall, for each tool, close to or below 50%.

Delaitre et al. [40] compiled results from SAMATE's competitions using the SARD [35] data set. They concluded that, in general, design and coding clarity helps improving software assurance, because the simpler the control and data flow structure of the test case, the more effective the tools will be at finding weaknesses [40]. They also concluded that distinct SCATs generally do not find the same weaknesses [40], due to differences in tool design.

Shiraishi, Mohan, and Marimuthu [49] proposed evaluation criteria and new test suites for benchmarking SCATs. After evaluating commercial SCATs, they concluded that there are defects difficult to detect. This general purpose test suite does not tackle cryptography at all.

Finally, Hoole at al. [41] analyzed test suites from SARD [35], discussed various deficiencies and associated improvements, evaluating such improvements against SCATs. They showed that invalid assumptions in test design lead to imprecise measurement.

## III. Overview of the Benchmarking Methodology

The methodology adopted in this work is based on that proposed by Antunes and Vieira [45]. We built our method around the concept of predefined, but realistic test cases [45]. The difference is that our test cases are specific for a whole domain (e.g., cryptography), instead of a single type of vulnerability in a broader domain. By coding test cases for types of bad practices found in an existing classification of cryptography misuse (described in Section III-B), we were able to capture the diversity of this domain. A **cryptography misuse** is a programming bad practice frequently found in cryptographic software [1], leading to vulnerabilities. We do not simply name them vulnerabilities because, in many cases, they are design flaws and insecure architectural choices.

The basic idea of our method is to exercise SCATs using sample programs for cryptography usage (with and without crypto misuses) and, based on the detected misuses, calculate a small set of metrics that capture the detection capabilities of SCATs in the cryptography domain. Also, we argue that, due to the many variables involved (e.g., crypto misuses, SCAT deployments, developer expertise in cryptography, the way misuses appear in applications, and availability of crypto experts), the definition of a benchmark for cryptography misuse detection tools must be specifically targeted to a particular scenario, in order to actually make choices about the benchmark components.

### A. Benchmarking Procedure

Figure 1 illustrates the proposed benchmarking procedure. Its four stages are then described in detail.

1) **Planning**, where the benchmark is scoped and designed, using knowledge about cryptography misuse, with the following steps:
    a) determine likely misuses of cryptography for certain application profiles (refer to Section IV);
    b) determine the development context, including team experience and expert availability, and their relation with base metrics (Section V);
    c) determine the benchmarking scenario using combinations of weighted metrics for SCATs, configured according to profiles and contexts (Section VI).
2) **Preparation** consists in identifying and configuring tools according to a selected scenario for conducting a benchmarking campaign (Section VII).
3) **Operation** (Section VII), where measures are collected to further analysis and comparison, with the following steps:
    a) execution of the tools for the specified tests, detecting misuses;

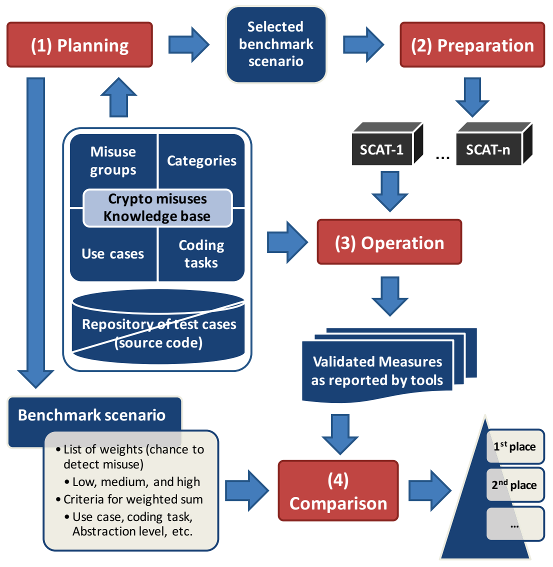
(1) Planning
Selected benchmark scenario
(2) Preparation
SCAT-1
...
SCAT-n
Misuse groups
Categories
Crypto misuses Knowledge base
Use cases
Coding tasks
Repository of test cases (source code)
(3) Operation
Validated Measures as reported by tools
Benchmark scenario
• List of weights (chance to detect misuse)
• Low, medium, and high
• Criteria for weighted sum
• Use case, coding task, Abstraction level, etc.
(4) Comparison
1st place
2nd place
...

Fig. 1. Overview of the benchmarking procedure.

   b) verification of raw measures (TP, FP, TN, and FN) by experts.
4) **Comparison** (Sections VII and VIII), where measurements are analyzed, interpreted, and prepared to presentation, with the following steps:
   a) normalization of results in order to compare the tools using common formats;
   b) computation of metrics from the raw measures;
   c) evaluation and ranking of the tools according to weighted metrics and scenarios.

The above procedure follows a general process for experimentation [50], customized for benchmarking of vulnerability detection tools [45]. The key novelty resides in the planning phase, which presents our contribution in systematically understanding how cryptography misuse affects development teams and tool selection.

### *B. Classification of Cryptography Misuses*

Braga and Dahab [1] proposed a classification of cryptography misuses to capture how software developers actually misuse cryptography in practice. Their classification has nine categories: Weak Cryptography (WC), Bad Randomness (BR), Coding and Implementation Bugs (CIB), Program Design Flaws (PDF), Improper Certificate Validation (ICV), Public-Key Cryptography (PKC) issues, Poor Key Management (PKM), Cryptography Architecture and Infrastructure (CAI) issues, and IV/Nonce Management (IVM) issues. Each category is further detailed in more descriptive subsets in Table II.

The classification was obtained by an iterative and incremental process. First, the authors analyzed in different case studies [2]–[4] the pitfalls developers have to avoid to properly use cryptography. They also reviewed selected literature for software security that cover cryptography issues [51]–[56], and adopted their labels, grouping them in the main categories with few sub-items. Then, new categories were added and sub-items were refined from industry sources [57]–[59] and recent papers [7]–[10], [12], [13], [60]. Validation and further refinement occurred by studying online communities for cryptography programming [1].

In this work, we extend that classification to include other misuses and apply it to categorize our test cases. The main additions are insecure elliptic curves [17], repeated nonce in ECDSA [16], as well as misconfiguration of RSA [14] and DH [15]. The resulting classification (in Table II) assembles cryptography misuse in software collected from various sources: literature on software security (e.g., [51]–[56]), recent studies on cryptography misuse (e.g, [7]–[10], [12], [13], [60]), newly discovered misuses in Java (e.g., [14]–[17]), and industry initiatives for software security (e.g., [57]–[59]). Table II shows the grouping of misuse categories, misuse main categories, subsets, and sources.

## IV. Application Profile and Misuse Groups

The application domain defines the cryptography requirements [5], which, in general, are satisfied by traditional use cases associated to cryptographic services [1], [18]: encrypting data at rest (EDR), secure communication (SC), password protection and encryption (PPE), authentication and validation of data (AVD), and digital rights management (DRM).

TABLE II
Crypto misuses from software security.

| MG | Misuse category | Misuse subtype | Misuse source |
|---|---|---|---|
| Misuse Group 1 (MG1) | Weak Cryptography (WC) | - Risky or broken crypto<br>- Proprietary cryptography<br>- Determin. symm. encryption<br>- Risky or broken hash/MAC<br>- Custom implementation | [7] [8] [9] [10] [60] [51] [52] [53] [54] [55] [57] [58] |
| | Coding and Implementation Bugs (CIB) | - Wrong configs for PBE<br>- Common coding errors<br>- Buggy IV generation<br>- Null cryptography<br>- Leak/Print of keys | [7] [8] [9] [60] [19] [54] [58] |
| | Bad Randomness (BR) | - Use of statistic PRNGs<br>- Predict., low entropy seeds<br>- Static, fixed seeds<br>- Reused seeds | [7] [8] [9] [60] [51] [52] [53] [54] [55] [57] [58] |
| Misuse Group 2 (MG2) | Program Design Flaws (PDF) | - Insecure default behavior<br>- Insecure key handling<br>- Insecure use of streamciphers<br>- Insecure combo encrypt/auth<br>- Insecure combo encrypt/hash<br>- Side-channel attacks | [7] [8] [10] [11] [60] [51] [52] [55] [56] [57] [58] |
| | Improper Certificate Validation (ICV) | - Missing validation of certs<br>- Broken SSL/TLS channel<br>- Incomplete cert. validation<br>- Improper validated host/user<br>- Wildcards, self-signed certs | [7] [10] [12] [13] [60] [56] [58] |
| | Public-Key Cryptography (PKC) issues | - Deterministic encrypt. RSA<br>- Insecure padding RSA enc.<br>- Weak configs for RSA enc.<br>- Insecure padding RSA sign.<br>- Weak signatures for RSA<br>- Weak signatures for ECDSA<br>- Key agreement: DH/ECDH<br>- ECC: insecure curves | [8] [9] [10] [11] [60] [14] [53] [55] [16] [17] |
| Misuse Group 3 (MG3) | IV and Nonce Management (IVM) issues | - CBC with non-random IV<br>- CTR with static counter<br>- Hard-coded or constant IV<br>- Reused nonce in encryption | [8] [9] [10] [60] |
| | Poor Key Management (PKM) | - Short key, improper key size<br>- Hard-coded or constant keys<br>- Hard-coded PBE passwords<br>- Reused keys in streamciphers<br>- Use of expired keys<br>- Key distribution issues | [7] [8] [9] [60] [51] [52] [53] [54] [55] [56] [59] [58] |
| | Crypto Architecture and Infrastructure (CAI) issues | - Crypto agility issues<br>- API misunderstanding<br>- Multiple access points<br>- Randomness source issues<br>- PKI and CA issues | [7] [8] [9] [10] [11] [13] [60] [51] [54] [55] [56] [57] [58] |

These use cases are implemented in crypto coding tasks [1], [18]: encryption and decryption (Enc/Dec), digital signatures and verification (Sig/Ver), hashes or authentication codes and verification (Hash/Mac), key generation or agreement (KA), secure channels (e.g., SSL), digital certificate validation (Cert), and randomness generation (Rand). There are also secondary tasks, accessory to the main ones, representing operational (but important) aspects of crypto systems: key distribution, certificate generation, and key storage and recovery. In general, every crypto use case can be accomplished by a combination of coding tasks, where the complexity of the task is determined by the actual use case at hand.

*Cryptography misuses* are introduced by developers during coding tasks [1]. For this reason, an assessment setup for evaluation of cryptography usage during software development has to evaluate misuses according to their impact on coding tasks and related use cases. Cryptography misuses are not all equally difficult to avoid: some are easier to find and correct than others, depending on the level of abstraction (coding, design, architecture) required to identify the misuse. In our experience and related work [1], there are three qualitative groupings for the nine misuse categories (in Table II):

**MG1** *(low complexity)* is related to coding activities and issues in APIs, and could be easily found by early detection techniques, simple code reviews, and skilled developers (supported by tools). It includes Weak Crypto (WC), Coding and Implementation Bugs (CIB), and Bad Randomness (BR). No deep understanding of program design is required, because fixes are likely to be simple and related to single programs or simple code snippets.

**MG2** *(medium complexity)* is related to flaws in program design affecting a few different programs and may be difficult to identify due to feature distribution across programs. It includes Improper Certificate Validation (ICV) issues, Program Design Flaws (PDF), and Public-Key Crypto (PKC) issues. Fixes may require program redesign and may affect a few programs. Avoiding them requires more knowledgeable developers and support from experts.

**MG3** *(high complexity)* is related to flaws in system design and architecture, and requires understanding of system architecture to analyze underlying cryptosystems. It includes Poor Key Management (PKM), IV and Nonce Management (IVM) issues, and Crypto Architecture and Infrastructure (CAI) issues. Fixes usually require new modules or redesign of modules, and may affect many code bases. These misuses require cryptography experts to perform code and design reviews, or architecture analysis.

*Application's Crypto Misuse Profile* determines likely crypto misuses in applications, as well as their complexity to be avoided by developers and likelihood to be detected by tools. A low-profile app benefits from early detection of misuses and is mainly associated to MG1; a high-profile app, associated with MG3, is better supported by late detection; and a medium-profile app, associated to MG2, is somewhere in between.

## V. Development Contexts and Base Metrics

The *development context* determines the availability of cryptographic knowledge to development teams, either as actual developers with applied cryptography knowledge or as external consultants. There are four combinations of two binary variables: team **experience** (with or without experience) and expert **availability** (available or unavailable), resulting in the development contexts listed in Table III. There is a recommended application profile and misuse groups for each context.

TABLE III
Contexts combining teams and application profiles.

| C# | Context | App profile | Misuse groups |
|---|---|---|---|
| C1 | Novice team, no expert | Low complexity | MG1 |
| C2 | Novice team with expert | Low to medium | MG1 and MG2 |
| C3 | Skilled team, no expert | Medium to high | MG2 and MG3 |
| C4 | Skilled team and expert | High complexity | MG3 |

Benchmarking metrics are selected according to development contexts and app profiles in which SCATs will be used. Specific combinations of these factors define a benchmarking scenario. Table IV summarizes the metrics suitable for each development context (further discussed later for each context). We adopt the following definitions for recall, precision, and f-measure [44]. Recall is the proportion of positive cases that are correctly classified as positive and is computed by the formula $TP/(TP + FN)$. Precision is the proportion of the classified positive cases that are correctly classified and is computed by the formula $TP/(TP + FP)$. F-Measure represents the harmonic mean of precision and recall, given by the simplified formula $2 * TP/(2 * TP + FN + FP)$. Next subsections describe, for each development context, how teams interact with crypto experts, how crypto misuse shows up, how SCATs can be used, and which metrics are better to evaluate SCATs.

TABLE IV
Development contexts and recommended metrics.

| Context | 1st. Metric | 2nd. Metric |
|---|---|---|
| C1 | Precision | Recall |
| C2 | Recall | Precision |
| C3 | F-Measure | Recall |
| C4 | Recall | Precision |

### A. Context 1: Unsupported Novice Team

This context comprises novice developers with little expert help, where novice developers want to avoid accessing (expensive or unavailable) crypto experts as much as possible, and solve simple issues by themselves. Accessing experts for false alarms produced by SCATs can be a waste of resources, but having expert help for double checking omissions is barely possible. When this novice team develops low-complexity apps, it may experience only simple misuses with code-based fixes, where no (or very few) program design is needed.

When this novice team develops medium-complexity apps, developers usually have to redesign a few programs when fixing misuses. In this case, SCATs are supposed to produce more false alarms. When the same novice team faces high-complexity apps (a barely recommended setup), developers usually suffer from complex misuses associated to system design and architecture. This context can result in software with the worst security (compared to other contexts) and is recommended only for low-complexity apps.

This way, a low false positive rate (FPR) is required for developers and a low false negative rate (FNR) helps optimize the time of experts. Also, precision is more important than recall, because experts will eventually be consulted later in verification and testing. This way, precision is the first metric for benchmarking SCATs and recall is a second option for tie breaking.

### B. Context 2: Supported Novice Team

In this context, novice developers have access to crypto experts and want to use them as much as possible to overcome team's lack of skill in cryptography. Thus, accessing experts for SCATs false alarms or omissions is not a big deal. For low-complexity apps, the team has expert help for solving false alarms and omissions. When the novice team develops medium-complexity apps, expert review can be applied to remove false positives. When the same novice team faces high-complexity apps, developers can rely solely on cryptography experts to solve complex misuses, and SCATs are supposed to support the expert's work.

In this context, experts help developers to learn applied cryptography. This can result in better security and is good for medium- to high-complexity apps, depending on the actual availability of experts. It is also recommended for developments with loose schedules and low time pressure. As experts can always help with false positives and false negatives, despite the waste of time and delays, recall is the first metric for benchmarking, and precision is tie tiebreaker.

### C. Context 3: Unsupported Knowledgeable Team

This comprises skilled developers acting as an autonomous team with rare expert help. In this context, experienced developers want to solve as many of the issues as possible, in a best effort approach, even if some issues are false positives. When faced with simple misuses, in low-complexity apps, this team is able to solve them quickly or recognize false alarms with a minimum waste of time.

On the other hand, when facing medium-complexity misuses, the time taken to discard false alarms may be relatively long, but still acceptable for a development cycle. In this setup, experts can be timely accessed to solve false alarms. In high-complexity apps, when complex misuses appear, developers may not be able to recognize all false alarms. Experts could quickly solve difficult issues, but they are not available for double checking omissions and false alarms. This context is suitable for medium-complexity apps. In this case, f-measure is the first metric and recall is the tiebreaker.

### D. Context 4: Supported Knowledgeable Team

This context comprises a team with both skilled developers and crypto experts. Developers and experts work together to solve as many issues as possible, even if some of them are false alarms and omissions. In both low- and medium-complexity misuses, this team is able to solve issues quickly and recognize false alarms with minimum delay. When complex misuses appear, developers may not recognize all false alarms, but with expert help they can quickly solve difficult issues within affordable delays. The expert time taken to discard omissions may be relatively long, but still acceptable. This context can result in the best security and is suitable for high-complexity apps. It is also recommended for developments with tight schedules and high time pressure.

In summary, experts and developers deal together with false positives and false negatives. Also, a high FPR can be acceptable, because expert support and budget are available to solve them. In fact, it is important to maximize recall, but precision is not as important, because team members can easily identify false positives and plan for review in search of false negatives. In this context, recall is the first metric and precision is the tiebreaker.

## VI. Benchmark Scenarios and Weighted Metrics

Benchmark scenarios define combinations of weighted metrics adopted to evaluate SCATs against likely misuses in development contexts. Like contexts, there are four main scenarios. The first scenario (S1) is a learner team with unavailable expert (C1) developing low-complexity apps, favoring simple misuses from MG1 (see Table III). The second (S2) is novice developers with highly available experts (C2), developing low- to medium-complexity apps, favoring MG1 and MG2. The third (S3) is an skilled team with unavailable expert (C3) that favors misuses from MG2 and MG3. The fourth scenario (S4) is skilled team with available experts (C4) that favors complex misuses from MG3. This section explains how the scenarios are built.

### A. Metrics and Weights

The workload represents the work to be executed by SCATs during evaluation and is given by a number of test cases actually exercised. In our method, when evaluating tools for a specific scenario, all available test cases are included in the workload, and weights determine the importance of each misuse for the benchmark scenario. Direct metrics are computed for each misuse category (and for each tool under evaluation). Then, weighted metrics are computed by a weighted sum of direct metrics.

We adopt a qualitative approach to determine weight, which is derived by the expected performance of tools in each scenario. A misuse has greater weight if tools are expected to detect it in a specific scenario. Weights are not related to operational risk, chance of exploitation, or expected loss.

Table V shows the weights for four scenarios. For instance, in scenario one (S1), a SCAT should be adequate for novice developers, without expert help, developing low-complexity apps (C1). This SCAT should prioritize the detection of crypto misuses in MG1. This way, weights are defined as high for MG1 and low for both MG2 and MG3.

Weighted metrics for misuse categories can be computed by the formula $\sum_{i=1}^{9} w(i) * m(i) / \sum_{i=1}^{9} w(i)$, where $w(i)$ is the weight for misuse category $i$ and $m(i)$ is the measured value for the metric in question (recall, precision, or f-measure). Weighted metrics for misuse groups, use cases, coding tasks, or any other criteria, can be computed accordingly. In a benchmark, it is necessary to choose a criteria. We adopted misuse categories for general tool evaluation, as well as misuse groups in our study case for tool benchmarking. Table VI shows four possible criteria to characterize misuses in a workload.

### B. Workload Characterization

In our method, workload selection is not a big deal since all test cases are always included in the workload. Characterizing the workload is the important factor influencing metrics.

The workload was derived directly from the classification work by writing programs to exemplify misuses and their variants, for

TABLE V
WEIGHTS FOR SCENARIOS, CONTEXT, AND MISUSE GROUPS.

| Scenario | Context | Weights per misuse group | | |
|---|---|---|---|---|
| | | MG1 | MG2 | MG3 |
| S1 | C1 | High | Low | Low |
| S2 | C2 | Medium | High | Low |
| S3 | C3 | Low | Medium | High |
| S4 | C4 | Low | Low | High |

TABLE VI
CRYPTO MISUSES DISTRIBUTED BY CATEGORIES.

| Criteria | Subset | Misuse | Good use |
|---|---|---|---|
| Misuse group | MG1 | 61 | 34 |
| | MG2 | 106 | 99 |
| | MG3 | 35 | 49 |
| Crypto use cases | EDR | 90 | 93 |
| | AVD | 49 | 39 |
| | RND | 13 | 10 |
| | PPE | 10 | 5 |
| | SC | 40 | 35 |
| Crypto coding tasks | Enc/Dec | 83 | 68 |
| | Sign/Ver | 26 | 27 |
| | Hash/MAC | 22 | 11 |
| | KG | 43 | 61 |
| | SSL | 10 | 3 |
| | Cert | 5 | 2 |
| | Rand | 12 | 10 |
| Misuse categories | WC | 20 | 10 |
| | CIB | 29 | 16 |
| | BR | 12 | 8 |
| | PDF | 23 | 14 |
| | ICV | 15 | 5 |
| | PKC/ENC | 27 | 30 |
| | PKC/SIG | 21 | 25 |
| | PKC/ECC | 14 | 20 |
| | PKC/KA | 6 | 5 |
| | IVM | 8 | 10 |
| | PKM | 19 | 32 |
| | CAI | 8 | 7 |

each misuse category and its sub-items. Currently, our workload consists of 202 misuses (positives) and 182 good uses (negatives), with a total of 384 test programs for Java and its crypto API [61]. Good uses are the negative cases in the confusion matrix, provided by the testing oracle (see table I). We added expert knowledge to our workload by tagging test cases with the following criteria (used to sum up weights when computing weighted metrics):

- Crypto use cases and coding tasks (e.g., [1], [18]);
- Misuse groups for contexts and app profiles;
- Misuse categories (see Table II);
- Positives and negatives (see Table VI);
- Target platform (only Java [61] by the time of writing).

In order to avoid a large number of test cases in public-key crypto (PKC) issues, 68 positives and 80 negatives, we divided this category in four subsets for encryption with RSA, digital signatures with RSA and ECDSA, elliptic curve cryptography (for insecure curve selection), and key agreement with DH and ECDH, in Table VI. The number of test cases, according to different criteria, are shown in Table VI. All source code is public:

https://bitbucket.org/alexmbraga/cryptomisuses
https://bitbucket.org/alexmbraga/cryptogooduses

## VII. ASSESSMENT OF SCATS FOR CRYPTOGRAPHY

This section describes the assessment of free SCATs with measures (TP, TN, FN, and FP) and metrics (precision, recall, and f-measure). We selected free SCATs from OWASP [62] and SAMATE [42]. By reading documentation, we identified five free SCATs for Java that also have rules for crypto issues: FindBugs 3.0.1 [63] with FindSecBugs 1.5.0 [27] (FSB), VisualCodeGrepper 2.1.0 [30] (VCG), Xanitizer 3.0.0 [29] (Xan), SonarQube 6.2 [28] with sonar-scanner 2.8 (SQ), and Yasca 3.0.5 [31]. All these tools perform late detection of vulnerabilities [32] and should be applied after developers have produced some source code.

Table VII puts together metrics for the five tools. They were computed uniformly for the workload (i.e., without considering the weights). Looking at precision and FP, a cursory glance would choose Yasca as the best tool, with higher precision and no FP. However, for the trained eye, Xanitizer would be a better choice, due to a higher recall and f-measure.

The tool with higher recall, Xanitizer (Xan), detected only one third of all misuses. The second higher recall (FSB) detected one quarter of misuses. Tools are not mutually exclusive and, in fact, have great intersection. When computing the union of TPs for all tools, we found a coverage of 35%, with 72 TPs (only 4 more than Xanitizer individually). These numbers are not good when compared to assessments with a broader scope [43], suggesting that free tools perform better in other security domains than in cryptography. Without a specific scenario to direct the comparison, f-measure seems to be the best metric to adopt, because it combines precision and recall. In this case, Xanitizer is the best choice.

TABLE VII
RESULTS FOR FIVE FREE SCATS.

| Tools | Measures | | | | Metrics | | |
|---|---|---|---|---|---|---|---|
| | TP | TN | FN | FP | Prec. | Recall | F-M |
| FSB | 51 | 147 | 151 | 35 | 0.593 | 0.252 | 0.354 |
| Xan | 68 | 140 | 134 | 42 | 0.618 | 0.337 | 0.436 |
| SQ | 5 | 181 | 197 | 1 | 0.833 | 0.025 | 0.048 |
| VCG | 7 | 180 | 195 | 2 | 0.778 | 0.035 | 0.066 |
| Yasca | 8 | 182 | 194 | 0 | 1.000 | 0.040 | 0.076 |

The following sections discuss the results in detail for misuse categories in the same abstraction level (MG1, MG2, and MG3).

### A. Assessment of SCATs for Misuse Group 1

Table VIII shows metrics MG1. Concerning weak cryptography (WC), Xan and FSB seem to be the best tools with most TPs, but their FPs are high, resulting in low precision. SQ, VCG, and Yasca have no FP, resulting in high precision. However, these tools show low recall. Xan has the larger coverage, followed by FSB. Xan wins in recall and f-measure, being the best tool in this category.

Most tools detected DES and 3DES as insecure. Yasca did not detect 3DES. FSB and Xan detected weak hash functions (e.g., MD2, MD4, MD5, and SHA-1). Yasca did not detect SHA-1. Only Xan detected indirect references to weak hash functions. Also, only Xan detected an insecure MAC based on SHA-1. FSB and Xan detected insecure ECB mode. No tool detected Blowfish, RC4 or insecure PBE. No tool detected manual compositions of RSA and hashes. Most evaluated

SCATs behaved as simple pattern matchers for strings, thus, unable to detect misuses dependent of data-flow analysis.

For coding and implementation bugs (CIB), three tools did not score (no TP nor FP): SQ, VCG, and Yasca. Xan had most TP, but many false alarms (FP), obtaining the largest values for precision, recall, and f-measure. FSB got a second place, mostly influenced by a low recall. Both Xan and FSB detected buggy IV generation and NullCipher. Only Xan detected leaks of private or secret keys. However, leaks of DH's shared secrets were not detected. Tools did not detect saved keys in strings, misconfigured PBE, and concatenated inputs for hashes. VCG looked for the word "password" in comments, but misses other languages.

For bad randomness (BR), Table VIII shows that FSB, Xan, VCG, and Yasca have the highest precision with no FPs. However, VCG and Yasca got low recall. SQ did not score. Interestingly, FSB and Xan are tied in all three metrics (no tiebreaker). Also, Xan and FSB detected all uses of statistic PRNG and VCG missed a few misuses. Xan, FSB, and VCG detected only a few cases of low-entropy seeds. No tool detected fixed seeds nor reuse of seeds, suggesting the existence of blind spots in evaluated tools.

## B. Assessment of SCATs for Misuse Group 2

Concerning program design flaws (PDF), Table IX shows that SQ, VCG, and Yasca did not score. FSB and Xan had the same value for recall, but FSB got the highest values for precision and f-measure. Taking precision as tiebreaker, FSB wins this category.

FSB and Xan detected insecure default for AES. FSB and Xan got insecure default for 3DES. Other insecure defaults (e.g., RSA, PBE, OAEP, and PRNG) were not detected. Tools have huge omissions in this category, showing many blind spots. No tool detected insecure combinations of encryption with hash or MAC. FSB and Xan detected some insecure stream ciphers. Side channels, such as timing channels in verification of hashes or MACs, and padding oracles, were not detected.

For public-key cryptography (PKC) issues, Table IX shows that VCG and Yasca did not score. FSB, Xan, and SQ got the highest precision with no FP. Xan wins this category due to higher recall and f-measure. Despite large TPs, the highest recall is still less than one quarter of all test cases for this category.

No tool detected insecure curves for ECC. Xan and FSB detected all cases of insecure padding for RSA and deterministic RSA. SQ got only one case. A bug in SQ makes it case sensitive for algorithm names, so that, for instance, two strings "RSA/NONE/NoPadding" and "RSA/None/NoPadding" are different and only the first one is detected. No tool detected insecure hashes or small parameters for RSA-OAEP. No tool detected small parameters for key agreement (e.g., DH or ECDH) and repeated message nonce for ECDSA. FSB and Xan detected only one case of weak configs for ECDSA (SHA1withECDSA), as well as only a few weak configs for RSA signatures. Tools are optimistic in this category to reduce FP, but having many omissions (FN). Also, tools are not updated with recent misuses, resulting in blind spots.

For improper certificate validation (ICV), according to Table IX, only FSB and Xan scored, with the same precision and no FP. FSB got better recall and f-measure for a higher TP, performing better in this category. Only FSB and Xan detected a few issues for certificate validation related to insecure SSL/TLS. FSB performed slightly better than Xan because of a bug in Xan that prevented the detection of misuses in nested classes. Again, tools limited themselves to just a few misuses in order to avoid mistakes.

## C. Assessment of SCATs for Misuse Group 3

Concerning IV and nonce management (IVM) issues (see Table X), FSB has higher recall, but medium precision. FSB also got a higher value for f-measure, being the best choice for this category. VCG, SQ, and Yasca did not score. FSB and Xan detected constant IVs. No tool detected non-random IV for CBC, nor static counters for CTR. FSB detected one case of nonce reuse. The high number of FPs indicates tools have difficulty in understanding program design for IV management.

In poor key management (PKM), FSB and Xan got the same scores in all measures, resulting in a hard tie in all three metrics, as shown in Table X. VCG, SQ, and Yasca did not score in this category. FSB and Xan detected the same few cases of constant keys, constant passwords for PBE, and key reuse in stream ciphers. In this case, tool builders were less optimistic and tried to capture possible, but uncertain issues, resulting in more FPs due to lack of program understanding. Key management is made of design decisions hard to get from code analysis.

Considering Crypto Architecture and Infrastructure (CAI) issues, insecure architectural decisions are the most difficult cryptography misuse to detect by looking only at code. Only Xanitizer scored in this category, obtaining relatively good results for derived metrics. This happened because of a small number of test cases for one issue detected only by Xanitizer: the omission of a crypto provider during algorithm selection. However, this may be an FP, because crypto providers can be defined in configuration files as well.

## VIII. Benchmarking Case Study

As an example, in this case study we select SCATs to be used by a novice team during development of an application with low-complexity crypto usage (context C1 in Table III). For illustrative purposes only, we suppose that the development process will target a hypothetical (but realistic) application for password protection and management, in which secure storage for passwords is encrypted by a master key derived from a master password. This simple app targets a quite common need in mobile devices [19]. This app has two main use cases: password protection with encryption (PPE) and Encrypting Data at Rest (EDR). Password integrity should also be guaranteed (AVD). A security assessment would show that its design and architecture are simple and key management is limited to a single master key/password. Development is likely to be affected by crypto misuses related to coding bugs (CIB), bad randomness usage (BR), password-based encryption misconfiguration (in CIB), and weak cryptography (WC).

The development team favors free tools and may only have part-time support of crypto experts, as external consultants. So we would like to select tools to promptly address MG1 issues, but also to point possible issues from MG2 and MG3, which will be eventually analyzed by an expert in the future. This way, the benchmark setup for this case study is that of scenario one (S1), consisting of development context one (C1 - Novice team, no expert), for a low-complexity app profile. In this setup, precision and recall are the adequate metrics, with recall as tiebreaker. Also, for computation of weighted metrics, misuse

TABLE VIII
RESULTS FOR MISUSE GROUP ONE (MG1).

| **Tools** | **Metrics for WC** | | | **Metrics for CIB** | | | **Metrics for BR** | | |
|---|---|---|---|---|---|---|---|---|---|
| | Prec. | Recall | F-M | Prec. | Recall | F-M | Prec. | Recall | F-M |
| FSB | 0.727 | 0.40 | 0.516 | 0.50 | 0.172 | 0.256 | 1.0 | 0.417 | 0.588 |
| Xan | 0.588 | 0.50 | 0.541 | 0.70 | 0.483 | 0.571 | 1.0 | 0.417 | 0.588 |
| SQ | 1.0 | 0.20 | 0.333 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| VCG | 1.0 | 0.20 | 0.333 | 0.0 | 0.0 | 0.0 | 1.0 | 0.250 | 0.400 |
| Yasca | 1.0 | 0.30 | 0.462 | 0.0 | 0.0 | 0.0 | 1.0 | 0.167 | 0.286 |

TABLE IX
RESULTS FOR MISUSE GROUP TWO (MG2).

| **Tools** | **Metrics for PDF** | | | **Metrics for PKC** | | | **Metrics for ICV** | | |
|---|---|---|---|---|---|---|---|---|---|
| | Prec. | Recall | F-M | Prec. | Recall | F-M | Prec. | Recall | F-M |
| FSB | 0.417 | 0.217 | 0.286 | 1.0 | 0.221 | 0.361 | 1.0 | 0.267 | 0.421 |
| Xan | 0.357 | 0.217 | 0.270 | 1.0 | 0.235 | 0.381 | 1.0 | 0.133 | 0.235 |
| SQ | 0.0 | 0.0 | 0.0 | 1.0 | 0.015 | 0.029 | 0.0 | 0.0 | 0.0 |

TABLE X
RESULTS FOR MISUSE GROUP THREE (MG3).

| **Tools** | **Metrics for IVM** | | | **Metrics for PKM** | | | **Metrics for CAI** | | |
|---|---|---|---|---|---|---|---|---|---|
| | Prec. | Recall | F-M | Prec. | Recall | F-M | Prec. | Recall | F-M |
| FSB | 0.286 | 0.500 | 0.364 | 0.263 | 0.263 | 0.263 | 0.0 | 0.0 | 0.0 |
| Xan | 0.231 | 0.375 | 0.286 | 0.263 | 0.263 | 0.263 | 0.800 | 1.0 | 0.889 |

TABLE XI
WEIGHTED METRICS FOR FIVE SCATS IN FOUR SCENARIOS.

| **Tools** | **Weighted metrics for S1** | | | **Weighted metrics for S2** | | | **Weighted metrics for S3** | | | **Weighted metrics for S4** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | W-Prec. | W-Recall | W-F-M | W-Prec. | W-Recall | W-F-M | W-Prec. | W-Recall | W-F-M | W-Prec. | W-Recall | W-F-M |
| Xan | **0.737** | 0.451 | 0.537 | 0.756 | **0.302** | 0.392 | 0.563 | 0.431 | **0.427** | 0.488 | **0.510** | 0.471 |
| FSB | 0.701 | 0.316 | 0.425 | 0.747 | 0.266 | 0.377 | 0.412 | 0.253 | 0.270 | 0.281 | 0.259 | 0.242 |
| Yasca | 0.556 | **0.130** | 0.208 | 0.208 | **0.049** | 0.078 | 0.042 | 0.010 | **0.016** | 0.056 | 0.013 | 0.021 |
| VCG | 0.556 | 0.125 | 0.204 | 0.208 | 0.047 | 0.076 | 0.042 | 0.009 | 0.015 | 0.056 | 0.013 | 0.020 |
| SQ | 0.306 | 0.056 | 0.093 | 0.313 | 0.024 | 0.041 | 0.125 | 0.006 | 0.010 | 0.056 | 0.006 | 0.010 |

groups MG1, MG2, and MG3 are weighted as high, low, and low, respectively (see Table V).

Table XI puts together weighted metrics for the five evaluated tools from Section VII in the four scenarios (sorted by scenario one, the one of interest in the current example). Weighted metrics were computed as weighted sums of metrics calculated for misuse categories and weights associated to misuse groups, as discussed before.

For scenario one, taking only weighted precision as reference, Xan is the best choice. By giving a higher weight to MG1, we actually amplified the importance of simple misuses, where Xanitizer scored slightly better than FSB and much better than other tools. Table XI shows three groups of scores for scenario one (S1): Xan and FSB with high scores, VCG and Yasca with medium scores, and SQ with a low score. These numbers indicate that Xan and FSB not only performed better in general, but also preferred simple misuses from MG1, confirming what we saw in Section VII: Xan and FSB (individually) have more rules for crypto than all other tools together and most of them are for MG1.

The unbalanced number of rules among SCATs favored Xan and FSB. Table XI shows that Xan and FSB easily ranked in first and second places in all scenarios. On the other hand, the competition for ranking in third place was not that ease. For scenario one (S1), Yasca got third place thanks to tie breaking in recall. In scenario two (S2), where recall is the first metric, Yasca performed better and got third place, despite being tied with VGC and performed worst than SQ in precision.

For scenario three (S3), where f-measure rules and recall is the tiebreaker, Yasca got third place with a tight difference from other two SCATs. For scenario four (S4), we found not only the smallest metrics in general, but also the smaller differences among weighted metrics for Yasca, VCG, and SQ. Yasca and VCG were tied in both preferred metrics for S4, so the contend was solved by f-measure in favor of Yasca with tine advantage.

Evaluated SCATs favor scenario one because of a technology bias found in free SCATs toward low-complexity misuses, as discussed in Section VII. When we forced the use of free SCATs in other scenarios, their performance reduces gradually as misuse complexity increases. This behavior is perceived in Table XI by the progressively reduced values of weighted metrics (from scenarios one to four).

## *A. Misuse Groups and Tool Behavior*

An optimistic tool only alerts about clear misuses and keeps silent about dubious ones. On the other hand, a pessimistic tool warns about every suspected misuse, even unlikely ones. SCATs for security have to be pessimistic in order to avoid dangerous omissions [53]. On the other hand, SCATs should favor early detection of vulnerabilities in order to benefit from developer's short-term memories when fixing vulnerable code [32].

Based upon our observations and findings, we generalized the following expected behaviors for crypto-friendly, pessimistic SCATs (Table XII). Tools can be quite precise in early detection of MG1 (WC, CIB, and BR), showing relatively few FPs. Recall inside the misuse group (group recall) is expected to be relatively high, with few FNs. However, non-detected misuses from MG2 and MG3 cause dangerous omissions (FNs in overall recall).

In MG2 (ICV, PDF, and PKC), tools are expected to be less precise, producing more false alarms and omissions than in MG1. This is expected due to FPs caused by incomplete understanding of program design, as well as FNs due to misconceptions about programs. In MG2, optimistic tools are expected to show relatively low recall and many FNs, while pessimistic tools are expected to have relatively low precision and many FPs. These tools are better suited to late detection of crypto misuses.

In MG3 (PKM, IVM, and CAI), tools are expected to be quite imprecise, producing more false alarms and omissions than in other misuse groups. This behaviour is expected because of FPs due to partial understanding of software architectures, as well as FNs due to misconceptions about program design. Optimistic tools will have relatively low recall, while pessimistic tools will have lower precision. Tools for MG3 are better suited to late detection of misuses.

As discussed previously, all free SCATs in this benchmark favor MG1 and gradually decrease their performance in (weighted) metrics for MG2 and MG3. Table XII summarizes the relation between misuse groups and tool behavior.

TABLE XII
EXPECTED TOOL BEHAVIOR FOR CRYPTO-FRIENDLY SCATS.

| Misuse group | Tool behavior |
|---|---|
| MG1 | Higher precision and high recall |
| MG2 | Low precision and moderate recall |
| MG3 | Lower precision and high recall |

## B. Use of SCATs for Cryptography

Knowing the limitations of tools is important to understand how to use them more effectively. Table XIII summarizes tool usages for misuse groups. First, the recommended usage of SCATs for MG1. By precisely detecting many code-based misuses, SCATs can be integrated to IDEs and provide real-time support to developers during coding tasks (e.g., early detection). Since precision is high, these tools will need less expert supervision in coding. Because overall recall is low, tools alone will result in low quality (less secure) software, which can be subjected to further verification. This tool usage is recommended for development context 1 (novice teams and barely available experts).

The recommended tool usage for MG2 follows. SCATs should not be the only way to find design flaws in coding, because design flaws are supposed to be found earlier in development, before coding tasks. If they are being found in coding, this may be a symptom of expert unavailability or team amateurism. Tools can be applied during system integration (daily or weekly build) and support design reviews or manual inspection, possibly later in system development. This tool usage is recommended for novice teams with expert help (context 2) or skilled teams and barely available experts (context 3).

As for MG3, SCATs should not be the only way to find architectural flaws in coding, because they are supposed to be found very early in development. If they are not, this may be a symptom of expert unavailability or team building issues (e.g., lack of security architects). Extensive use of tools during coding tasks is inadequate and can mistakenly divert team effort to correct nonexistent misuses or correct existing, but complex, misuses the wrong way. This tool usage is better for supporting experts during manual inspections in architectural security analysis and is recommended for development contexts where crypto experts are always available (e.g., contexts two and four).

Considering the evaluated tools, FindSecBugs can be integrated to IDEs and building tools. VisualCodeGrepper (VCG) has both CLI and GUI, and its CLI can be called by scripts in building processes, but not integrated to IDEs. Xanitizer can be used as a standalone tool with its own GUI and also be incorporated in a build process. SonarQube can be integrated to building tools and claims to have plugins for many IDEs.

Finally, the current version of the best ranked tool in this benchmark (Xanitizer) is not easily integrated to IDEs. The second option (FindSecBugs) is better suited for late detection of misuses within IDEs. Also, Xanitizer and FindSecBugs are adequate for contexts one and two, with the drawback of showing relatively more false positives and false negatives in the second case. We do not believe any of evaluated tools is adequate for contexts three and four, due to their poor performance in this benchmark.

TABLE XIII
CONTEXTS LINK LIKELY MISUSES AND TOOL USAGE.

| Context | Misuse group | Usage | Tool |
|---|---|---|---|
| C1 | MG1 | Integrated to IDE | Xan and FSB |
| C2 | MG1 and MG2 | IDE and build | Xan and FSB |
| C3 | MG2 and MG3 | Build and review | None |
| C4 | MG3 | Reviews | None |

## IX. DISCUSSION AND THREATS TO VALIDITY

This section discusses limitations that threat validity, and other general aspects of our benchmarking methodology.

### A. Limitations and Threats to Validity

We are aware that our workload is a sample of all possible test cases for cryptography misuse. For instance, we did not implement all variations of algorithm names in case-sensitive strings. Also, we explicitly avoided test cases for specific technologies and platforms (e.g., web, mobile, etc.).

We found a general coverage of around 35%, with the two best scores around 33% and 25%. These numbers are not surprising. For instance, Antunes and Vieira [45] concluded in their benchmark that the effectiveness of tools is poor. Also, OWASP's benchmark [43] of tools for web security shows a coverage of 50% for the best tool. Other works [38], [39] have similar results. These arguments suggest that tools perform better in other security domains than in cryptography.

Most evaluated tools use pattern matching as the main technique for vulnerability detection. In particular, VisualCodeGrepper, SonarQube, and Yasca seem to use only simple pattern matching. FindSecBugs detects some non-trivial patterns, such as zeroed IVs and insecure interface implementations. Xanitizer uses taint analysis to detect simple leaks of keys and indirect references to weak algorithms.

Our workload is made of realistic test cases, meaning that they are not artificial, but they are not real applications either. They were collected and derived from crypto misuse patterns found in many sources. Still, they may contain crypto misuses not frequently found in real applications. On the other hand, the diversity provided by our test cases is hard to obtain by only using actual applications.

When loading test cases into tools, auxiliary code had to be excluded from the workload. Otherwise, evaluated tools would consider them in vulnerability scans, generating even more false positives. A simple and fast solution is to exclude auxiliary code from raw measures.

Our test cases target crypto misuse and neglect other aspects of code quality (including secure coding) in order to keep programs small, self-contained, and self-explained. However, these design decisions were not captured by tools. Thus, when validating raw measures, we had to exclude issues not related to crypto misuses.

Crypto misuses can appear in combinations of two or three, showing a strong relation to each other [1]. Because of that, some test cases had to include more than one misuse, in order to capture the intended misbehavior. In these cases, we chose to ignore auxiliary misuses (when detected by tools) and focus on detection of main misuses. Auxiliary misuses have their own test cases.

When evaluated tools had no default configuration that included verification of crypto misuses, we always preferred the more complete (full) scan. Also, for output, we preferred reports with all severity classes (levels) enabled. This practice was adopted to avoid complicated settings and customized configurations, which can be hard to reproduce.

### *B. General Discussion*

Cryptography misuse is not related to the implementation of cryptographic algorithms. Instead, crypto misuses emerge when ordinary developers use cryptographic libraries (APIs) in their daily coding activities. These developers can also use SCATs to receive detailed reports about issues and how to correct them.

This work sits in the intersection of three domains (e.g., applied cryptography, software engineering, and software security) sharing a common knowledge base. We believe that our experiment actually tests what we meant to test, because tool builders have codified this knowledge (to some degree) in SCATs, which are able to detect cryptography misuse, even in limited ways.

SCATs generate detailed reports that have to be manually inspected. By inspecting these reports, we were able to identify all issues related to cryptography, separating them from other issues, and classifying them as true positives and false positives. We mitigated the threat of mislabeling misuses, because crypto misuses were clearly identifiable due to the systematic use of our classification to generate test cases and our experience in this kind of code review.

Finally, our conclusions are based upon a repeatable experiment and are consistent with related work and our previous results. In fact, our numbers were consistent among executions, because evaluated SCATs are deterministic and our benchmarking method is reproducible. Previous results were obtained from case studies, an extensive literature review, and empirical studies of online communities.

## X. Conclusion

Cryptography usage is full of design decisions that defy early detection of misuse. Also, coverage of cryptography misuse by tools is far from good, with current tools showing many blind spots. We saw a huge gap between what experts actually see as cryptography misuse and what tools currently detect. Therefore, expert help is required to assure quality in different moments of development efforts. We argue that, with current tools' maturity, an adequate toolkit has to be carefully crafted to fit the needs of teams for specific development contexts.

Our methodology was able to distinguish among tools for distinct usage scenarios and find the best suited options, despite limitations in current SCAT technology. Also, the method provided the necessary context to analyze the reasons why evaluated tools are not suitable for advanced scenarios.

As future work, in the short term, we plan to improve test cases to include new misuses, variations of existing ones, as well as perform assessments of commercial-off-the-shelf tools. In the long run, we plan to expand test cases to other programming languages, software platforms, cryptographic libraries, as well as dynamic analysis tools. Finally, we foresee the emergence of a new generation of tools able to better assist the development of secure cryptographic software.

## Acknowledgment

This work has been partially supported by CNPq, Intel, and the projects **DEVASSES** (www.devasses.eu), funded by the European Union's FP7 under grant agreement no. PIRSES-GA-2013-612569, and **EUBra-BIGSEA** (www.eubra-bigsea.eu), funded by the European Commission under Horizon 2020 grant agreement no. 690116. R. Dahab and A. Braga acknowledge the support of Fapesp's thematic project #2013/25.977-7. Authors also thank UNICAMP, CPqD, and University of Coimbra for the institutional support.

## References

[1] A. Braga and R. Dahab, "Mining Cryptography Misuse in Online Forums," in *2nd International Workshop on Human and Social Aspect of Software Quality*, 2016.

[2] A. Braga and D. Schwab, "Design Issues in the Construction of a Cryptographically Secure Instant Message Service for Android Smartphones," in *The 8th Inter. Conf. on Emerging Security Information, Systems and Technologies*, nov 2014, pp. 7–13.

[3] A. Braga, R. Zanco Neto, A. Vannucci, and R. Hiramatsu, "Implementation Issues in the Construction of an Application Framework for Secure SMS Messages on Android Smartphones," in *The 9th Inter. Conf. on Emerging Security Information, Systems and Technologies*. IARIA, 2015, pp. 67–73.

[4] A. Braga and A. Colito, "Adding Secure Deletion to an Encrypted File System on Android Smartphones," in *The 8th Inter. Conf. on Emerging Security Information, Systems and Technologies*, nov 2014, pp. 106–110.

[5] A. Braga and R. Dahab, "Towards a Methodology for the Development of Secure Cryptographic Software," in *The 2nd International Conference on Software Security and Assurance (ICSSA 2016)*, 2016.

[6] ——, "A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software," in *XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2015)*, Florianópolis, SC, Brazil, 2015, pp. 30–43.

[7] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why Does Cryptographic Software Fail?: A Case Study and Open Problems," in *5th Asia-Pacific Workshop on Systems*, ser. APSys '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:7.

[8] A. Chatzikonstantinou, C. Ntantogian, C. Xenakis, and G. Karopoulos, "Evaluation of Cryptography Usage in Android Applications," *9th EAI International Conference on Bio-inspired Information and Communications Technologies*, 2015.

[9] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," *ACM SIGSAC conference on Computer & comm. security (CCS'13)*, pp. 73–84, 2013.

[10] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications," in *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC)*, 2014, pp. 75–80.

[11] P. Gutmann, "Lessons Learned in Implementing and Deploying Crypto Software," *Usenix Security Symposium*, 2002.

[12] M. Georgiev, S. Iyengar, and S. Jana, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, 2012, pp. 38–49.

[13] S. Fahl, M. Harbach, and T. Muders, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in *ACM conference on Computer and communications security*, 2012, pp. 50–61.

[14] E. S. Alashwali, "Cryptographic vulnerabilities in real-life web servers," in *Third International Conference on Communications and Information Technology (ICCIT)*. IEEE, jun 2013, pp. 6–11.

[15] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, and Others, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 5–17.

[16] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow, "Elliptic curve cryptography in practice," in *Financial Cryptography and Data Security*. Springer, 2014, pp. 157–175.

[17] V. G. Mart and L. Hern, "Implementing ECC with Java Standard Edition 7," *International Journal of Computer Science and Artificial Intelligence*, vol. 3, no. 4, pp. 134–142, 2013.

[18] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, ""Jumping Through Hoops": Why do Java Developers Struggle With Cryptography APIs?" *The 38th International Conference on Software Engineering*, 2016.

[19] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Analysis on Password Protection in Android Applications," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on*, nov 2014, pp. 504–507.

[20] S. Fahl, M. Harbach, and H. Perl, "Rethinking SSL development in an appified world," *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pp. 49–60, 2013.

[21] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, and M. Mezini, "Towards Secure Integration of Cryptographic Software," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2015)*. New York, NY, USA: ACM, 2015, pp. 1–13.

[22] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting SSL usage in applications with SSLint," in *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 519–534.

[23] OWASP, "OWASP Testing Project," 2015. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Project

[24] J. Rizzo and T. Duong, "Practical padding oracle attacks," *Proc. of the 4th USENIX conf. on offensive technologies (2010)*, pp. 1–9, 2010.

[25] OWASP, "OWASP Top Ten Project," 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10

[26] SANS/CWE, "TOP 25 Most Dangerous Software Errors." [Online]. Available: www.sans.org/top25-software-errors

[27] P. Arteau, "FindSecBugs." [Online]. Available: https://find-sec-bugs.github.io

[28] SonarSource, "SonarQube." [Online]. Available: https://www.sonarqube.org

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

[30] NCCGroup, "VisualCodeGrepper." [Online]. Available: https://github.com/nccgroup/VCG

[31] M. Scovetta, "Yasca." [Online]. Available: http://yasca.org

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

[33] Y. Li, Y. Zhang, J. Li, and D. Gu, "iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications," in *8th International Conference on Network and System Security*. Xi'an, China: Springer International Publishing, 2014, pp. 349–362.

[34] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.

[35] NIST, "Software Assurance Reference Dataset (SARD)." [Online]. Available: https://samate.nist.gov/SRD/

[36] P. E. Black, "Counting bugs is harder than you think," in *11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2011, pp. 1–9.

[37] P. Black, "Static analyzers: Seat belts for your code," *IEEE Security & Privacy*, vol. 10, no. 3, pp. 48–52, 2012.

[38] G. Díaz and J. R. Bermejo, "Static analysis of source code security: Assessment of tools against SAMATE tests," *Information and Software Technology*, vol. 55, no. 8, pp. 1462–1476, 2013.

[39] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, vol. 68, pp. 18–33, 2015.

[40] A. Delaitre, B. Stivalet, E. Fong, and V. Okun, "Evaluating bug finders - Test and measurement of static code analyzers," *Proceedings - 1st International Workshop on Complex Faults and Failures in Large Software Systems, COUFLESS 2015*, pp. 14–20, 2015.

[41] A. M. Hoole, I. Traore, A. Delaitre, and C. de Oliveira, "Improving Vulnerability Detection Measurement: [Test Suites and Software Security Assurance]," *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pp. 27:1–27:10, 2016.

[42] NIST, "Software Assurance Metrics And Tool Evaluation." [Online]. Available: https://samate.nist.gov

[43] OWASP, "OWASP Benchmark Project." [Online]. Available: https://www.owasp.org/index.php/OWASP_Benchmark_Project

[44] N. Antunes and M. Vieira, "On the Metrics for Benchmarking Vulnerability Detection Tools," in *The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*. Rio de Janeiro, Brazil: IEEE, 2015.

[45] ——, "Assessing and Comparing Vulnerability Detection Tools for Web Services: Benchmarking Approach and Examples," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 269–283, 2015.

[46] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[47] J. A. Kupsch and B. P. Miller, "Manual vs. automated vulnerability assessment: A case study," in *First International Workshop on Managing Insider Security Threats (MIST)*, 2009, pp. 83–97.

[48] L. Manohar, R. Velicheti, D. C. Feiock, M. Peiris, R. Raje, and J. H. Hill, "Towards Modeling the Behavior of Static Code Analysis Tools," *2014 9th Cyber and Information Security Research Conference*, 2014.

[49] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 12–15, 2015.

[50] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[51] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, 2001.

[52] M. Howard and D. LeBlanc, *Writing Secure Code*, 2nd ed. Redmond, Wash: Microsoft Press, Dec. 2004.

[53] B. Chess and J. West, *Secure programming with static analysis*, 2007.

[54] M. Howard, D. LeBlanc, and J. Viega, *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill Education, 2009.

[55] M. Howard and S. Lipner, *The Security Development Lifecycle*. Redmond, WA, USA: Microsoft Press, 2006.

[56] A. Shostack, *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[57] Safecode, "Fundamental Practices for Secure Software Development," 2011. [Online]. Available: http://www.safecode.org/wp-content/uploads/2014/09/SAFECode_Dev_Practices0211.pdf

[58] OWASP, "Cryptographic Storage Cheat Sheet." [Online]. Available: www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

[59] CYBSI, "Avoiding The Top 10 Software Security Design Flaws," 2014. [Online]. Available: http://cybersecurity.IEEE.org/

[60] A. Braga and R. Dahab, "Introdução à Criptografia para Programadores: Evitando Maus Usos da Criptografia em Sistemas de Software," in *Caderno de minicursos do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015*, 2015, pp. 1–50.

[61] Oracle, "Java Cryptography Architecture (JCA) Reference Guide." [Online]. Available: docs.oracle.com/javase/8/docs/technotes/guides/\security/crypto/CryptoSpec.html

[62] OWASP, "List of Source Code Analysis Tools." [Online]. Available: https://www.owasp.org/index.php/Source_Code_Analysis_Tools

[63] UMD, "FindBugs." [Online]. Available: http://findbugs.sourceforge.net