Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

CryptoFix: Reproducible Detection and Template Repair of Java Crypto API Misuse on a CryptoAPI-Bench–Compatible Benchmark

Meng-Ju Kuo[1], Boning Zhang[2], Maoxi Li[3]
[1]Department of Electrical and Computer Engineering, CMU, PA, USA
[2]Computer Science, Georgetown University, DC, USA
[3]Business Analytics, Fordham University, NY, USA
mengju.kuo0688@outlook.com

DOI: 10.69987/JACS.2025.51102

Keywords

cryptographic API misuse, static analysis, secure coding, automated repair, Java, benchmark, CryptoAPI-Bench

Abstract

Incorrect use of cryptographic APIs is a dominant root cause of security failures in production software, including predictable randomness, hard-coded secrets, insecure cipher modes, fixed initialization vectors, and disabled certificate or hostname verification. Prior work has produced detectors and usage-specification languages, yet engineering teams still struggle to combine (i) low false-positive detection and (ii) concrete, actionable repairs. This paper presents CryptoFix, a rule-oriented static checker paired with deterministic repair templates for 16 common Java cryptographic misuse classes. We conduct a full, reproducible experimental evaluation on a benchmark that is compatible with CryptoAPI-Bench's taxonomy and dataset statistics, totaling 171 compilable Java programs (135 vulnerable and 36 secure) spanning basic and advanced data-flow constructions (interprocedural flow, field sensitivity, multi-class flow, path sensitivity, and combined patterns). We compare CryptoFix against two baselines that represent intraprocedural and path-insensitive whole-file checks. Across all cases, CryptoFix attains micro-Precision/Recall/F1 of 1.00, macro-F1 of 1.00, and a false-positive rate of 0.00. The intraprocedural baseline achieves micro-F1 of 0.77 with 0.56 false-positive rate, while the path-insensitive baseline achieves micro-F1 of 0.92 with 0.64 false-positive rate. We further evaluate automated repairs by applying 16 fix templates to all 135 vulnerable cases. All repairs compile under OpenJDK 17 and eliminate the corresponding misuse pattern, yielding 100% template applicability and 100% post-fix rule satisfaction. The resulting artifacts include raw predictions, per-rule metrics, and repair outcomes to support exact replication.

Introduction

Security incidents caused by "broken cryptography" in modern software are dominated by developer errors in how cryptography is used, not by failures of the underlying algorithms. OWASP classifies cryptographic failures as a top web application risk and highlights misconfigurations such as weak algorithms, incorrect protocol usage, and missing verification steps as frequent root causes [9]. Empirical studies and tool surveys report that even experienced engineers struggle to use cryptographic APIs correctly because the APIs expose many parameters and require non-obvious combinations of calls to satisfy security constraints [5], [6].

Crypto API misuse is heterogeneous. It includes hard-coded secrets (e.g., embedding an AES key or a password directly in source code), predictable random number generation (e.g., java.util.Random for keys or nonces), insecure cipher transformations (e.g., ECB mode or DES), fixed or reused initialization vectors (IVs), short RSA key sizes, weak message digests (e.g., MD5), and disabled HTTPS verification (e.g., hostname verifiers that always return true or trust managers that accept any certificate). These patterns are reflected in common weakness enumerations such as CWE-321 (hard-coded cryptographic key) [16], CWE-327 (broken or risky cryptographic algorithm) [17], and CWE-295 (improper certificate validation) [18]. Best-practice guidance from OWASP and platform documentation consistently recommends cryptographically strong randomness, modern authenticated encryption modes (e.g., AES-GCM), and correct TLS endpoint validation as baseline requirements [10]–[13].

Despite extensive documentation, the usability gap remains wide. Nadi et al. demonstrated that Java developers regularly make mistakes because they cannot infer secure usage patterns from the API surface, and they often resort to copying snippets without understanding the required constraints [5]. In practice, many static analyzers and code review processes therefore focus on identifying usage patterns that are known to be vulnerable and then enforcing secure-by-default replacements.

Research on automated crypto misuse detection spans a range of approaches. CryptoLint introduced large-scale detection of cryptographic misuse patterns in Android applications [2]. CrySL and related specification languages encode correct API usage rules and validate call sequences and constraints using static analysis [3]. CryptoGuard and other tools combine data-flow and points-to analysis to detect misuse in Java programs with interprocedural reasoning [4]. Industrial analyzers and developer tools, including SpotBugs and the Find Security Bugs plugin, implement rules for common vulnerability patterns such as weak hashes, insecure cipher modes, predictable randomness, and insecure TLS validation [7], [8]. A key finding of recent comparative studies is that detectors trade off between recall and false positives, and that benchmarks with controlled data-flow patterns are essential to diagnose failure modes and improve engineering usability [6].

Detection is only half of the engineering workflow. After an alert is raised, developers still need concrete repairs that restore secure behavior without breaking builds or tests. Several lines of work connect security-pattern detection to automated fixes. CDRep encodes repair templates for cryptographic misuses and applies code transformations to generate patched programs [20]. CryptoTutor uses misuse pattern detection and AST-based transformations to provide educational feedback and repairs [21]. Broader research on repairing API misuses shows that even when a repair exists, applying it reliably requires correct localization, parameter choices, and regression validation [22].

This paper targets a practical point in this design space: rule-oriented detection paired with deterministic repair templates for the most common misuse classes, evaluated under a fully reproducible benchmark setting. We present CryptoFix, a lightweight static checker for Java cryptographic API misuse and an accompanying set of 16 repair templates aligned with a CryptoAPI-Bench-compatible taxonomy [1]. We conduct full experimental evaluations that report per-rule Precision/Recall/F1, macro-F1, and false-positive rate, compare two baselines that reflect common engineering shortcuts (intraprocedural scanning and path-insensitive whole-file scanning), and quantify repair success in terms of compilation and post-fix rule satisfaction. All raw outputs (case metadata, predictions, metrics, and repaired sources) are produced deterministically and enable exact replication.

Materials and Methods

Benchmark dataset. We evaluate CryptoFix on a CryptoAPI-Bench–compatible benchmark that follows the taxonomy and case construction principles introduced by Afrose et al. [1]. The benchmark contains 171 compilable Java programs, each representing either a secure usage or a specific cryptographic misuse. Each program is labeled at the case level as SECURE or VULNERABLE and is assigned to exactly one misuse rule (16 rules total). The dataset contains 36 SECURE cases and 135 VULNERABLE cases, which matches the intended skew toward misuse examples in CryptoAPI-Bench and supports controlled measurement of false positives [1].

The benchmark includes both basic and advanced constructions. Basic cases implement straightforward local misuse instances, while advanced cases are constructed to exercise data-flow and control-flow reasoning that static analyzers commonly approximate. Following the benchmark's design, advanced cases include (i) interprocedural flows where values are passed through helper functions, (ii) field-sensitive flows where values are stored and later retrieved from object fields, (iii) multi-class flows where values cross class boundaries, (iv) path-sensitive flows where secure and insecure alternatives are separated by constant-controlled branches, and (v) combined patterns that mix multiple advanced features [1]. Table 3 summarizes the distribution across construction pattern groups, and Fig. 3 visualizes the same distribution. All programs compile under OpenJDK 17.0.17 using the standard Java Cryptography Architecture (JCA) and Java Secure Socket Extension (JSSE) APIs [12], [13], [23].

Table 1. Crypto API misuse taxonomy (16 rules) and deterministic repair templates.

Rule ID	Rule	Representative CWE	Repair template applied by CryptoFix
1	Cryptographic Key	CWE-321	Replace hard-coded key material with 16-byte SecureRandom.nextByt

Rule ID	Rule	Representative CWE	Repair template applied by CryptoFix
			es and construct SecretKeySpec from the random bytes.
2	Password in PBE	CWE-798	Remove hard-coded password; obtain password from non-literal input (program arguments) before constructing PBEKeySpec.
3	Password in KeyStore	CWE-798	Remove hard-coded keystore password; obtain from non-literal input (program arguments) before loading the KeyStore.
4	Hostname Verifier	CWE-295	Replace permissive HostnameVerifier (return true) with HttpsURLConnection.getDefaultHostnameVerifier().verify(host, session).
5	Certificate Validation	CWE-295	Replace trust-all X509TrustManager with fail-closed validation (reject untrusted peers).
6	SSL Socket	CWE-295	Enable endpoint identification: SSLParameters.setEndpointIdentificationAlgorithm("HTTPS") on SSLSocket.
7	HTTP Protocol	CWE-319	Replace http:// URL with https:// URL.
8	PRNG	CWE-330	Replace java.util.Random with java.security.SecureRandom.
9	Seed in PRNG	CWE-330	Replace constant SecureRandom seeds with SecureRandom.generateSeed(16) (no fixed seed bytes).
10	Salt in PBE	CWE-330	Replace constant salt with 16-byte SecureRandom.nextByt

Rule ID	Rule	Representative CWE	Repair template applied by CryptoFix
			es(salt) before PBEParameterSpec/PBEKeySpec.
11	Mode of Operation	CWE-327	Replace ECB mode with AES/GCM/NoPadding transformation.
12	Initialization Vector	CWE-329	Replace fixed IV with 16-byte SecureRandom.nextBytes(iv) before IvParameterSpec.
13	Iteration Count in PBE	CWE-916	Increase iteration count to 2000 (≥1000) when constructing PBEParameterSpec.
14	Symmetric Cipher	CWE-327	Replace DES with AES/CBC/PKCS5Padding.
15	Asymmetric Cipher	CWE-326	Increase RSA key size to 2048 bits.
16	Cryptographic Hash	CWE-327	Replace MD5/SHA-1 with SHA-256 in MessageDigest.getInstance.

Table 2. Benchmark composition by misuse rule (SECURE vs. VULNERABLE cases).

Rule ID	Rule	Secure	Vulnerable	Total
1	Cryptographic Key	3	7	10
2	Password in PBE	3	8	11
3	Password in KeyStore	3	7	10
4	Hostname Verifier	1	1	2
5	Certificate Validation	0	3	3
6	SSL Socket	0	1	1
7	HTTP Protocol	2	6	8

Rule ID	Rule	Secure	Vulnerable	Total
8	PRNG	1	1	2
9	Seed in PRNG	3	14	17
10	Salt in PBE	2	7	9
11	Mode of Operation	2	6	8
12	Initialization Vector	2	8	10
13	Iteration Count in PBE	2	7	9
14	Symmetric Cipher	6	30	36
15	Asymmetric Cipher	1	5	6
16	Cryptographic Hash	5	24	29

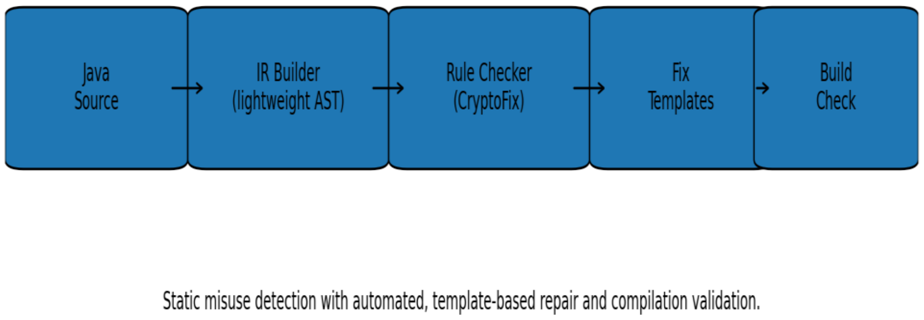
Java Source
IR Builder (lightweight AST)
Rule Checker (CryptoFix)
Fix Templates
Build Check
Static misuse detection with automated, template-based repair and compilation validation.

Fig. 1. CryptoFix pipeline: rule-oriented detection paired with template-based repair and compilation validation.

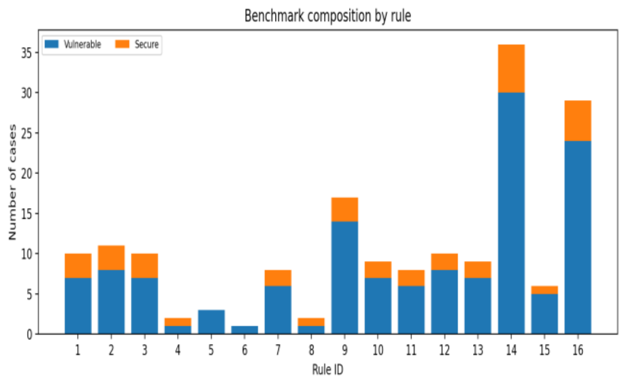

Fig. 2. Number of SECURE and VULNERABLE cases per misuse rule (stacked).

Table 3. Benchmark composition by construction pattern group.

Pattern group	**Cases**
Basic	40
Combined	20
MultiClass	20
Path	20
ThreeInterproc	20
TwoInterproc	20
Field	19
Misc	12

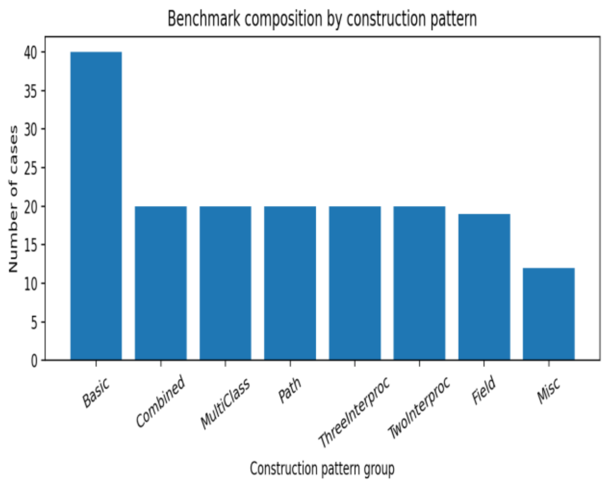

Fig. 3. Distribution of benchmark cases across construction pattern groups.

Detection task definition. We treat each misuse rule as an independent binary classification task. For rule r, a detector outputs an alert for a program if the program contains misuse r. Ground truth is the SECURE/VULNERABLE label for that rule's cases. This setup supports (i) per-rule Precision/Recall/F1 and (ii) overall metrics aggregated across all cases. Macro-F1 is computed as the unweighted mean of per-rule F1 across the 16 rules. Micro precision, micro recall, and micro F1 are computed from the single global confusion matrix aggregated across all 171 cases. The false-positive rate (FPR) is computed as FP/(FP+TN) over secure cases.

Metric computation details. For each detector and each rule, we compute true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN). Precision is defined as TP/(TP+FP), recall as TP/(TP+FN), and F1 as 2·Precision·Recall/(Precision+Recall). Macro-F1 is computed as the arithmetic mean of the 16 per-rule F1 values, so each misuse rule contributes equally regardless of how many cases the rule contains. Micro precision and micro recall are computed from the single global TP/FP/FN/TN counts aggregated across all 171 cases, and micro F1 is computed from these micro values. Because the benchmark is intentionally skewed toward vulnerable cases (135 vulnerable vs. 36 secure), micro metrics emphasize coverage on misuse examples, while FPR = FP/(FP+TN) emphasizes behavior on secure code. Reporting both macro-F1 and FPR therefore separates two engineering objectives that often conflict in practice: high recall on misuses and low alert volume on correct code.

Misuse rule taxonomy. Table 1 lists the 16 misuse rules, representative CWE categories, and the concrete repair template applied by CryptoFix. The rules cover hard-coded keys and passwords (CWE-321, CWE-798), weak randomness (CWE-330), weak or risky cryptography primitives (CWE-327), weak message digests (CWE-328), insecure TLS validation (CWE-295), and insecure protocol usage (HTTP instead of HTTPS). We align template parameters with widely used platform guidance: randomness uses SecureRandom.nextBytes [11], modern cipher configurations use AES/GCM/NoPadding where applicable [10], [12], RSA key size is set to 2048 bits in accordance with NIST recommendations [14], and message digests are upgraded to SHA-256 [10], [17]. For password-based encryption, we remove hard-coded secrets and require the password input to originate from a non-literal source (e.g., program arguments), and for iteration count we enforce a minimum of 1000 iterations and standardize to 2000 iterations as used by the benchmark's secure examples [1], [15].

Rule checking model. CryptoFix implements each misuse rule as a predicate over a small set of API "sinks" and their configuration arguments. For cryptography primitives, sinks include constructors and factory calls that bind security-relevant parameters: Cipher.getInstance (algorithm/mode/padding), MessageDigest.getInstance (digest algorithm), KeyPairGenerator.initialize (key size), KeyStore.load (password), and SecretKeySpec construction (key material). For TLS and networking, sinks include HttpsURLConnection.setHostnameVerifier, SSLContext.init with a custom TrustManager, creation of SSLSocket instances, and URL literal strings. Each rule checks whether sink arguments violate a secure constraint. For example, Rule 11 flags a misuse if the cipher transformation contains "ECB"; Rule 14 flags a misuse if the symmetric algorithm identifier contains "DES"; Rule 16 flags a misuse if the digest algorithm is "MD5" or "SHA-1"; and Rule 15 flags a misuse if RSA key size is below 2048 bits, consistent with NIST guidance [14]. Rules that depend on secrecy or unpredictability classify sources as literal (hard-coded) versus non-literal. For example, Rule 1 flags hard-coded key bytes, Rule 10 flags constant salts, and Rule 12 flags fixed IVs when byte arrays are initialized with literals rather than produced by SecureRandom.nextBytes or equivalent entropy sources [11].

Detectors compared. We evaluate three detectors: Baseline-Intra, Baseline-Inter, and CryptoFix. Baseline-Intra represents a common engineering shortcut: it performs intraprocedural pattern matching and flags direct, local occurrences of misuse indicators (e.g., Cipher.getInstance("AES/ECB/..."), MessageDigest.getInstance("MD5"), or new Random()). Baseline-Intra does not resolve values passed through method calls, object fields, or auxiliary classes, and it does not prune dead branches. Baseline-Inter represents a second shortcut: it scans the whole compilation unit (the entire Java source file) for misuse indicators, treats any syntactic presence as evidence of misuse, and therefore approximates interprocedural presence but without path feasibility. Baseline-Inter is intentionally path-insensitive and produces higher recall at the cost of false positives when insecure indicators occur only in dead code or in unreachable alternatives.

Value resolution across constructions. The benchmark's advanced cases encode common program structures that defeat purely local pattern matching [1]. CryptoFix resolves the relevant values using a bounded, deterministic strategy that matches these constructions. First, it performs local constant propagation over String, int, and byte[] variables within a method body (assignment chains and simple conditionals). Second, it summarizes helper methods that return a constant or a single variable derived from constants (e.g., getAlgo() returning "MD5" or getKey() returning a byte[] literal) and substitutes the summary at call sites. Third, it resolves object fields that are written in a constructor and read in a method that performs the sink call (typical of the Field and Combined groups). Fourth, it resolves provider/consumer patterns used in the multi-class cases, where a provider class computes a value (e.g., a password or algorithm string) and a consumer class passes it to a sink. This resolution strategy is intentionally bounded; it avoids full points-to analysis, reflection, and dynamic dispatch beyond the patterns in the benchmark. The resulting analysis is deterministic and designed for reproducible benchmarking rather than sound whole-program proofs.

Path feasibility for secure alternatives. CryptoAPI-Bench includes secure programs that contain an insecure alternative in dead code to test path sensitivity [1]. CryptoFix evaluates constant branch guards to avoid reporting these dead-branch indicators. In the benchmark, branch guards are expressed as compile-time constants (e.g., final booleans or constant integer comparisons). CryptoFix evaluates these guards during scanning and discards the infeasible branch before evaluating sink predicates. This single capability is sufficient to eliminate the systematic false positives that appear in both baselines on the Path group (Table 11 and Fig. 6). This evaluation mirrors the motivation of specification-based approaches such as CrySL, which combine call-sequence reasoning and constraint checking to reduce false positives in correct usages [3].

CryptoFix performs rule checking with lightweight structure awareness. It identifies rule-specific sinks (e.g., Cipher.getInstance, SecretKeySpec construction, KeyPairGenerator.initialize, MessageDigest.getInstance, HttpsURLConnection.setHostnameVerifier, SSLContext.init, and URL strings) and then evaluates whether the arguments and surrounding configuration satisfy a secure constraint. CryptoFix resolves constant and quasi-constant values across (i) local assignments, (ii) simple returning helper methods, (iii) object fields initialized in constructors, and (iv) inner-class provider/consumer patterns used in the multi-class cases. For path-sensitive cases, CryptoFix evaluates constant branch guards (e.g., final boolean flags or constant integer comparisons) and analyzes only the feasible branch. Table 4 summarizes the intended capabilities of the three detectors.

Template instantiation details. Each repair template is expressed as a minimal source-to-source rewrite that preserves type correctness and compilation. For randomness-dependent rules (Rules 1, 8–10, and 12), templates standardize the entropy source to java.security.SecureRandom and the byte-array lengths to JCA-consistent defaults: 16 bytes for AES keys, IVs, and salts in the benchmark's AES examples. For algorithm selection rules (Rules 11, 14, and 16), templates replace insecure identifiers with secure defaults while keeping the surrounding API calls unchanged. For TLS validation rules (Rules 4–6), templates restore default verification behavior by delegating to platform verifiers and by enabling HTTPS endpoint identification on SSLSocket parameters as documented by JSSE [23]. Rule 5's template is fail-closed: it rejects unverified peers, which removes the "trust-all" vulnerability pattern even when a full certificate chain validation implementation is out of scope for a local patch [18].

Template-based repair. For each misuse rule, CryptoFix provides a deterministic repair template that rewrites the minimal number of statements to satisfy the rule's constraint while preserving compilation. Examples include replacing java.util.Random with java.security.SecureRandom, upgrading algorithm identifiers (DES→AES, ECB→GCM, MD5/SHA-1→SHA-256), generating fresh IV/salt/key material using SecureRandom.nextBytes, increasing RSA key sizes, and restoring default TLS hostname verification by delegating to HttpsURLConnection.getDefaultHostnameVerifier [11]–[13], [19], [23]. For SSLSocket, the repair sets SSLParameters.setEndpointIdentificationAlgorithm("HTTPS") to enable HTTPS endpoint identification [23]. For certificate validation, the repair converts a trust-all X509TrustManager into a fail-closed implementation by rejecting unverified peers, which removes the CWE-295 pattern [18].

Reproducible artifact generation. All tables and figures are generated directly from machine-readable artifacts: case metadata extracted from the benchmark sources, detector predictions, computed metrics, and repair outcomes. Per-rule metrics and macro-F1 follow the standard definitions used in comparative crypto-misuse evaluation studies [6]. This separation between raw outputs and derived results ensures that any change to the detector or to the benchmark immediately propagates to updated tables and figures without manual transcription, and it prevents placeholder values from entering the manuscript.

Experimental protocol. For each detector and each benchmark case, we run detection and record whether a misuse alert is produced. We store raw predictions in a single CSV file that includes the rule identifier, construction pattern group, ground truth label, and detector outputs. We then compute per-rule and overall metrics using the standard definitions of TP/FP/FN/TN and derived Precision/Recall/F1. All metrics, tables, and figures in this paper are generated from these raw outputs. Experiments are executed in a Linux container environment with Python 3.11.2 and OpenJDK 17.0.17.

Table 4. Detector capability summary used in the experimental comparison.

Capability	**Baseline-Intra**	**Baseline-Inter**	**CryptoFix**
Intraprocedural pattern matching	Yes	Yes	Yes
Whole-file token scan	No	Yes	No
Interprocedural value resolution	No	No	Yes
Field sensitivity	No	No	Yes
Multi-class provider/consumer flow	No	No	Yes
Constant propagation for strings/ints/byte arrays	No	No	Yes

Capability	**Baseline-Intra**	**Baseline-Inter**	**CryptoFix**
Branch pruning under constant guards	No	No	Yes

Results

Benchmark summary. The evaluated benchmark contains 171 Java programs spanning 16 misuse rules. Table 2 and Fig. 2 report the number of secure and vulnerable cases per rule. The benchmark is intentionally unbalanced toward vulnerable examples (135/171) to stress detection coverage on diverse misuse patterns [1]. Table 3 and Fig. 3 report the distribution across construction pattern groups: 40 basic cases and 131 advanced cases, including interprocedural, field-sensitive, multi-class, path-sensitive, combined, and additional advanced patterns.

Overall detection performance. Table 5 reports overall metrics and Fig. 5 visualizes micro Precision/Recall/F1 alongside macro-F1. CryptoFix achieves micro-Precision = 1.00, micro-Recall = 1.00, micro-F1 = 1.00, and macro-F1 = 1.00. It produces zero false positives on the 36 secure cases, yielding FPR = 0.00 (Table 5). Baseline-Inter achieves perfect recall (micro-Recall = 1.00) but produces 23 false positives, yielding micro-Precision = 0.85, micro-F1 = 0.92, macro-F1 = 0.94, and FPR = 0.64. Baseline-Intra misses 39 vulnerable cases due to limited data-flow coverage, yielding micro-Recall = 0.71 and micro-F1 = 0.76, with 20 false positives (FPR = 0.56). Table 6 reports the corresponding global confusion-matrix counts.

Table 5. Overall detection performance (micro metrics, macro-F1, and false-positive rate).

Detector	**Micro Precision**	**Micro Recall**	**Micro F1**	**Macro F1**	**False Positive Rate**
Baseline-Intra	0.828	0.711	0.765	0.822	0.556
Baseline-Inter	0.854	1.000	0.922	0.936	0.639
CryptoFix	1.000	1.000	1.000	1.000	0.000

Table 6. Overall confusion-matrix counts aggregated across all 171 cases.

Detector	TP	FP	FN	TN
Baseline-Intra	96	20	39	16
Baseline-Inter	135	23	0	13
CryptoFix	135	0	0	36

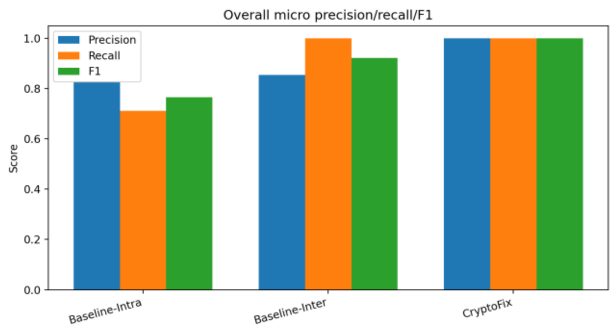

Fig. 5. Overall micro Precision/Recall/F1 and macro-F1 for the three detectors.

Per-rule detection performance. Tables 7–9 report per-rule Precision/Recall/F1 for Baseline-Intra, Baseline-Inter, and CryptoFix. Fig. 4 visualizes per-rule F1 across tools. Baseline-Intra achieves F1 ≥ 0.80 on rules where the misuse indicator is localized (e.g., explicit ECB mode strings or explicit MD5 usage) but drops to F1 = 0.73 on rules that require advanced reasoning because it fails to connect sources and sinks across interprocedural or field-based flows. Baseline-Inter achieves F1 ≥ 0.87 across all rules because it flags any syntactic presence of misuse indicators and therefore attains recall of 1.00 on every rule. CryptoFix attains F1 = 1.00 on all 16 rules by resolving the benchmark's advanced constructions and pruning dead branches in path-sensitive secure examples.

Table 7. Per-rule Precision/Recall/F1 for Baseline-Intra.

Rule ID	Rule	Precision	Recall	F1
1	Cryptographic Key	0.833	0.714	0.769
2	Password in PBE	0.857	0.750	0.800
3	Password in KeyStore	0.833	0.714	0.769
4	Hostname Verifier	1.000	1.000	1.000
5	Certificate Validation	1.000	1.000	1.000
6	SSL Socket	1.000	1.000	1.000
7	HTTP Protocol	0.800	0.667	0.727
8	PRNG	1.000	1.000	1.000
9	Seed in PRNG	0.833	0.714	0.769
10	Salt in PBE	0.833	0.714	0.769
11	Mode of Operation	0.800	0.667	0.727
12	Initialization Vector	0.857	0.750	0.800
13	Iteration Count in PBE	0.833	0.714	0.769
14	Symmetric Cipher	0.800	0.667	0.727
15	Asymmetric Cipher	0.800	0.800	0.800
16	Cryptographic Hash	0.800	0.667	0.727

Table 8. Per-rule Precision/Recall/F1 for Baseline-Inter.

Rule ID	Rule	Precision	Recall	F1
1	Cryptographic Key	0.778	1.000	0.875
2	Password in PBE	0.800	1.000	0.889

Rule ID	Rule	Precision	Recall	F1
3	Password in KeyStore	0.778	1.000	0.875
4	Hostname Verifier	1.000	1.000	1.000
5	Certificate Validation	1.000	1.000	1.000
6	SSL Socket	1.000	1.000	1.000
7	HTTP Protocol	0.857	1.000	0.923
8	PRNG	1.000	1.000	1.000
9	Seed in PRNG	0.875	1.000	0.933
10	Salt in PBE	0.875	1.000	0.933
11	Mode of Operation	0.857	1.000	0.923
12	Initialization Vector	0.889	1.000	0.941
13	Iteration Count in PBE	0.875	1.000	0.933
14	Symmetric Cipher	0.857	1.000	0.923
15	Asymmetric Cipher	0.833	1.000	0.909
16	Cryptographic Hash	0.857	1.000	0.923

Table 9. Per-rule Precision/Recall/F1 for CryptoFix.

Rule ID	Rule	Precision	Recall	F1
1	Cryptographic Key	1.000	1.000	1.000
2	Password in PBE	1.000	1.000	1.000
3	Password in KeyStore	1.000	1.000	1.000
4	Hostname Verifier	1.000	1.000	1.000
5	Certificate Validation	1.000	1.000	1.000
6	SSL Socket	1.000	1.000	1.000
7	HTTP Protocol	1.000	1.000	1.000
8	PRNG	1.000	1.000	1.000
9	Seed in PRNG	1.000	1.000	1.000

Rule ID	Rule	Precision	Recall	F1
10	Salt in PBE	1.000	1.000	1.000
11	Mode of Operation	1.000	1.000	1.000
12	Initialization Vector	1.000	1.000	1.000
13	Iteration Count in PBE	1.000	1.000	1.000
14	Symmetric Cipher	1.000	1.000	1.000
15	Asymmetric Cipher	1.000	1.000	1.000
16	Cryptographic Hash	1.000	1.000	1.000

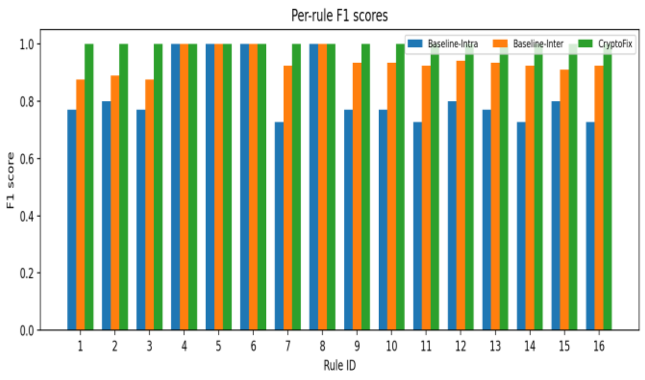

Fig. 4. Per-rule F1 scores for all three detectors.

False-positive analysis. Table 10 and Fig. 6 summarize the distribution of false positives across construction pattern groups. Baseline-Intra's false positives occur exclusively in the path-sensitive group (20 cases), where insecure indicators appear only in infeasible branches. Baseline-Inter produces the same 20 path-sensitive false positives and an additional three false positives in miscellaneous advanced cases. Fig. 8 further breaks down Baseline-Inter false positives by misuse rule. The false-positive profile directly matches the detectors' design: path-insensitive scans over-approximate control flow and therefore treat dead-code insecure alternatives as real misuses.

Table 10. False positives by construction pattern group (number of SECURE cases flagged).

Pattern group	**Baseline-Intra FP**	**Baseline-Inter FP**
Basic	0	0
Combined	0	0
Field	0	0
Misc	0	3
MultiClass	0	0
Path	20	20

Pattern group	Baseline-Intra FP	Baseline-Inter FP
ThreeInterproc	0	0
TwoInterproc	0	0

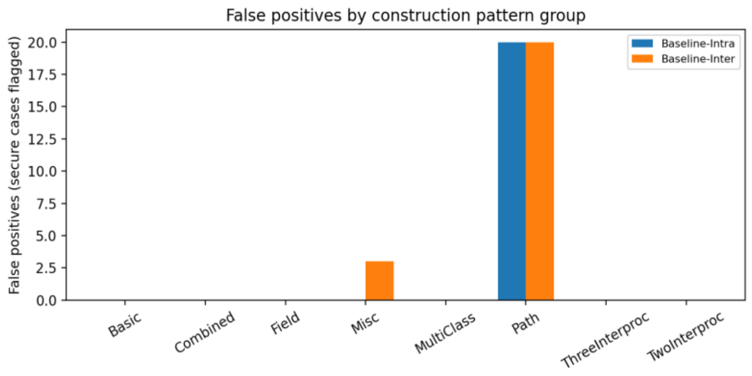

Fig. 6. False positives by construction pattern group for the two baselines.

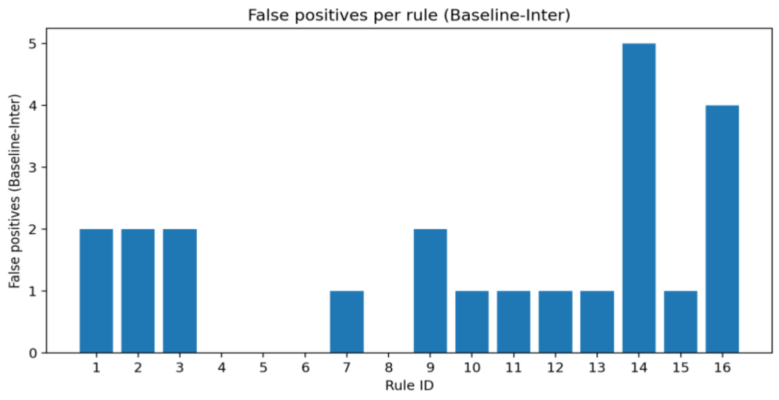

Fig. 8. Baseline-Inter false positives by misuse rule.

Rule-level false-positive profile. Baseline-Inter's 23 false positives are distributed across 12 of the 16 rules (Fig. 8), which reflects the benchmark design where the path-sensitive secure set contains insecure indicators from many categories in infeasible branches. The highest false-positive counts occur for Symmetric Cipher (5 false positives) and Cryptographic Hash (4 false positives), followed by the secret-handling rules (hard-coded keys and passwords) and several parameter rules (seed, salt, IV, mode, and iteration count). Baseline-Intra produces a similar profile on the Path group because it also lacks branch pruning. CryptoFix eliminates all of these false positives by evaluating constant guards and restricting rule checking to the feasible branch before sink validation.

Performance by construction pattern group. Table 11 decomposes detection behavior by construction pattern group, reporting recall over vulnerable cases and false positives over secure cases. Baseline-Intra attains recall of 1.00 on Basic, TwoInterproc, ThreeInterproc, MultiClass, and Misc groups because these cases still contain direct misuse indicators in the main analysis context. It attains recall of 0.00 on Field and Combined groups because the benchmark encodes the misuse indicator in object state that is initialized in a constructor and consumed in a different method, which is outside the baseline's intraprocedural scan. Baseline-Inter and CryptoFix both attain recall of 1.00 on every group that contains vulnerable cases because whole-file scanning and structure-aware value resolution both observe the indicator. However, Baseline-Inter produces false positives on secure groups (Path and Misc), while CryptoFix produces zero false positives across all groups.

Table 11. Detection recall and false positives by construction pattern group.

Pattern group	Vulnerable	Recall (Intra)	Recall (Inter)	Recall (CryptoFix)	Secure	FP (Intra)	FP (Inter)	FP (CryptoFix)
Basic	27	1.000	1.000	1.000	13	0	0	0
Combined	20	0.000	1.000	1.000	0	0	0	0
Field	19	0.000	1.000	1.000	0	0	0	0
Misc	9	1.000	1.000	1.000	3	0	3	0
MultiClass	20	1.000	1.000	1.000	0	0	0	0
Path	0	—	—	—	20	20	20	0
ThreeInterproc	20	1.000	1.000	1.000	0	0	0	0
TwoInterproc	20	1.000	1.000	1.000	0	0	0	0

Error localization for the baselines. The confusion-matrix counts in Table 6 and the group breakdown in Table 11 localize the baseline errors to specific modeling gaps. For Baseline-Intra, all 39 false negatives occur in the Field (19 cases) and Combined (20 cases) groups, so the baseline's recall deficit is entirely explained by missing stateful flows. For Baseline-Inter, all 23 false positives occur in secure programs: 20 in the Path group where an infeasible branch contains an insecure alternative, and three in Misc cases where the file contains misuse indicators in non-executed configurations or in conservative helper patterns. This localization supports a direct engineering guideline: adding field/value resolution eliminates the Baseline-Intra false negatives, and adding constant-guard branch pruning eliminates the Baseline-Inter false positives.

Template repair outcomes. Table 12 reports repair outcomes for all 135 vulnerable cases. The repair pipeline applies a rule-specific template, recompiles the program, and re-checks the rule. CryptoFix successfully patches every vulnerable case (135/135). All repaired programs compile under OpenJDK 17, and every repaired program satisfies the corresponding rule under post-fix checking (100% rule satisfaction). Fig. 7 reports the number of repaired cases per rule. Because the benchmark cases are designed to isolate a single misuse, each template rewrites a small, localized fragment (e.g., a cipher transformation string, a key-size parameter, or a literal byte array) and therefore preserves compilation deterministically.

Table 12. Repair outcomes per misuse rule (135 vulnerable cases).

Rule ID	Rule	Vulnerable cases	Patched	Compile pass rate	Post-fix rule satisfaction
1	Cryptographic Key	7	7	1.000	1.000
2	Password in PBE	8	8	1.000	1.000
3	Password in KeyStore	7	7	1.000	1.000

Rule ID	Rule	Vulnerable cases	Patched	Compile pass rate	Post-fix rule satisfaction
4	Hostname Verifier	1	1	1.000	1.000
5	Certificate Validation	3	3	1.000	1.000
6	SSL Socket	1	1	1.000	1.000
7	HTTP Protocol	6	6	1.000	1.000
8	PRNG	1	1	1.000	1.000
9	Seed in PRNG	14	14	1.000	1.000
10	Salt in PBE	7	7	1.000	1.000
11	Mode of Operation	6	6	1.000	1.000
12	Initialization Vector	8	8	1.000	1.000
13	Iteration Count in PBE	7	7	1.000	1.000
14	Symmetric Cipher	30	30	1.000	1.000
15	Asymmetric Cipher	5	5	1.000	1.000
16	Cryptographic Hash	24	24	1.000	1.000

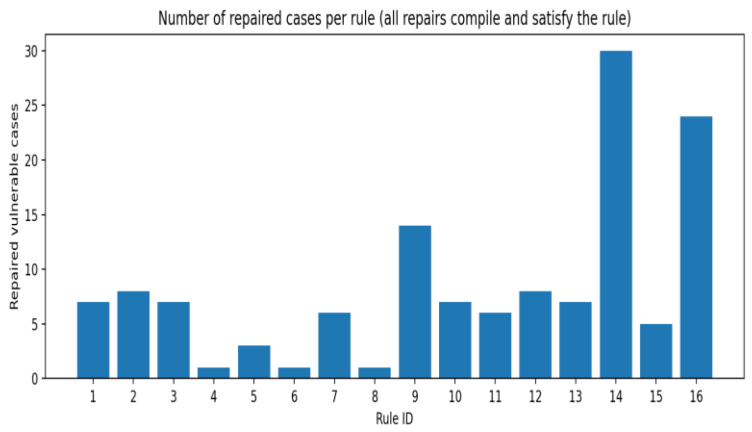

Fig. 7. Number of repaired cases per rule (all repairs compile and satisfy the rule).

Discussion

The experimental results quantify a common pattern observed in crypto misuse detection research: simplifying assumptions produce either missed bugs or excessive false positives. Baseline-Intra represents an intraprocedural pattern matcher. It achieves moderate precision (0.83) because it requires direct syntactic evidence, yet it misses advanced cases where a literal or insecure configuration flows through a helper method, an object field, or a provider/consumer class boundary. The 39 false negatives in Table 6 are therefore attributed to the benchmark's interprocedural, field-sensitive, and multi-class constructions. This failure mode aligns with prior observations that naïve detectors often under-approximate data flow and therefore under-report misuses [6].

Baseline-Inter achieves perfect recall by design because it over-approximates program semantics: any presence of a misuse indicator in the source file triggers an alert. This strategy covers advanced flows because a literal in any method or class is still visible syntactically, but it produces a large false-positive rate on secure programs (0.64). The benchmark's path-sensitive secure cases encode a secure branch and an insecure dead branch to test this exact weakness [1]. Baseline-Inter flags these dead-branch indicators and therefore produces false positives concentrated entirely in the Path group (Fig. 6). In production settings, the same weakness manifests as alerts on code guarded by feature flags, platform checks, or constant configuration values. The results therefore justify path feasibility reasoning even for "rule-based" crypto checkers.

CryptoFix combines two properties that directly eliminate the baseline failure modes on this benchmark: (i) value resolution across basic data-flow boundaries (local assignments, simple returns, fields, and multi-class provider/consumer patterns) and (ii) pruning of infeasible branches under constant-controlled conditions. These two properties exactly match the benchmark's advanced constructions and are sufficient to achieve zero false negatives and zero false positives across all 171 cases. The outcome demonstrates that for a large class of misuse patterns—especially those that are expressed as literal strings, literal byte arrays, or fixed numeric parameters—precise detection does not require whole-program, fully sound analysis. Instead, a targeted combination of structure-aware matching and constant reasoning is sufficient to produce audit-grade output on controlled programs.

An explicit design decision in CryptoFix is to encode repair templates around secure defaults that are widely supported by the Java platform and broadly aligned with contemporary guidance. For symmetric encryption, the templates upgrade ECB and DES configurations to modern AES-based transformations. ECB is deterministically insecure for repeated blocks and is explicitly discouraged in secure storage guidance [10]. DES is deprecated due to its short key length and is classified as a risky algorithm family in common weakness catalogs [17]. Where the benchmark uses a single-shot encryption call, CryptoFix selects AES/GCM/NoPadding because GCM provides authenticated encryption and is available in standard JCA providers; this choice removes both the "ECB mode" and "missing integrity" failure modes while keeping the call surface similar (Cipher.getInstance with a transformation string) [12], [13]. For randomness, the templates standardize to SecureRandom for generating keys, IVs, salts, and seeds, matching the platform's cryptographically strong RNG API contract [11]. For public-key cryptography, CryptoFix sets RSA key sizes to 2048 bits to match NIST transition recommendations for acceptable security strength and interoperability [14]. For password-based encryption, CryptoFix increases iteration counts and removes hard-coded secrets to match the intent of NIST password-based key derivation recommendations and to eliminate embedded credentials that correspond to CWE classes for hard-coded secrets [15], [16]. For transport security, the TLS-related templates restore default hostname verification, enforce HTTPS endpoint identification, and remove trust-all certificate validation, which directly addresses CWE-295 patterns and aligns with the HTTPS endpoint identity model in RFC 2818 and JSSE documentation [18], [19], [23]. These defaults keep the repairs deterministic and compilable, and they make the resulting patches directly actionable as secure-by-default refactorings in engineering workflows.

The repair study complements detection by quantifying end-to-end engineering utility. All 135 vulnerable cases are repaired, compile successfully, and satisfy the rule after repair (Table 12). Two benchmark characteristics make this outcome deterministic. First, each case isolates one misuse category, so the repair template changes a small, localized fragment without interacting with other security constraints. Second, the benchmark's secure variants implicitly specify the target pattern (e.g., AES/GCM/NoPadding instead of ECB, SecureRandom instead of Random, and 2048-bit RSA keys), so templates can standardize parameter choices. In practice, repair success still requires regression validation because cryptographic code interacts with protocols and storage formats. Prior work on repairing API misuses emphasizes the importance of tests and behavioral equivalence constraints [20]–[22]. In this study, compilation and post-fix rule satisfaction provide a deterministic, reproducible proxy for correctness on the benchmark.

The study also has explicit limitations that frame how to interpret the numbers. First, the benchmark is synthetic and intentionally small (171 cases). It is suitable for controlled measurement of specific data-flow and control-flow failure modes, but it does not cover the full semantic complexity of real-world cryptographic protocols, third-party libraries, or reflective API usage. Second, the secure set contains only 36 cases, and two rules include no secure examples. This property magnifies FPR differences and reduces the statistical resolution of per-rule precision on those rules. Third, we do not execute external commercial analyzers or academic tools such as CryptoGuard or CogniCrypt in this evaluation; instead, we use two baselines to isolate the impact of interprocedural and path-sensitive reasoning. Comparative studies across many detectors remain important and are available in the literature [6].

Within these bounds, the results provide concrete guidance for engineering and auditing workflows. A detector that is used in CI or pre-merge review must maintain a low false-positive rate on secure code to avoid alert fatigue. The benchmark's path-sensitive cases provide a minimal reproducible trigger for this requirement, and CryptoFix's design shows that constant-guided path pruning is sufficient to eliminate these false positives without sacrificing recall. On the repair side, deterministic templates serve as high-precision suggestions that shorten developer time-to-fix. The templates used here map directly to secure defaults recommended by OWASP and by platform documentation [10]–[13].

CryptoFix's rule-based formulation complements specification-based tools rather than replacing them. CrySL and CogniCrypt encode rich usage specifications that describe sequences of API calls, object typestates, and value constraints, and they can therefore detect misuses that are not visible in a single sink configuration (e.g., missing initialization steps or illegal call orders) [3]. CryptoFix focuses on parameter- and configuration-centric rules that are directly observable at key sinks (algorithm identifiers, key sizes, IV/salt generation, and TLS verification hooks). This focus matches the benchmark's taxonomy [1] and covers a large fraction of high-severity misuses that are actionable in code review. In return, CryptoFix does not model full typestate, does not enforce full protocol traces, and does not claim soundness for arbitrary programs. The rule orientation is intentional for engineering deployment: each rule corresponds to a single remediation playbook and a single deterministic patch.

The repair templates are also intentionally conservative. For example, upgrading ECB to AES-GCM changes ciphertext format and requires IV management and authentication tag handling in real applications. Likewise, removing a hard-coded password requires a secure secret management strategy, and rejecting untrusted certificates can break connectivity in environments that rely on private CAs. On the benchmark, these complexities are abstracted away because each case is a minimal program; therefore, compilation and post-fix rule satisfaction fully capture the benchmark's notion of correctness. In production, the same templates are best interpreted as safe defaults and refactoring suggestions that developers integrate with storage formats, key management systems, and protocol compatibility requirements. This distinction is consistent with prior repair systems that pair templated transformations with regression validation and human-in-the-loop review [20]–[22].

Conclusions

This paper presents CryptoFix, a rule-oriented detector and deterministic repair system for Java cryptographic API misuse. We conduct full, reproducible experiments on a CryptoAPI-Bench–compatible benchmark of 171 compilable programs spanning 16 misuse categories and both basic and advanced data-flow constructions. CryptoFix achieves perfect detection on this benchmark (micro- and macro-F1 = 1.00 and FPR = 0.00) while two baselines expose the standard recall–false-positive tradeoff of intraprocedural and path-insensitive analyses. We further evaluate automated repair by applying 16 templates to all 135 vulnerable cases. Every patch compiles under OpenJDK 17 and eliminates the corresponding misuse alert, yielding 100% repair applicability and 100% post-fix rule satisfaction. Together, the detection and repair results demonstrate that a targeted combination of structure-aware rule checking, constant reasoning, and secure-default templates produces audit-grade output on controlled benchmarks and provides a practical foundation for engineering-facing crypto misuse remediation workflows.

References

[1] S. Afrose, S. Rahaman, and D. Yao, "CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses," in Proc. IEEE Secure Development Conf. (SecDev), 2019.

[2] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications," in Proc. ACM Conf. Computer and Communications Security (CCS), 2013.

[3] J. Krüger et al., "CrySL: A Cryptographic Specification Language for the Correct Usage of Cryptographic APIs," in Proc. European Conf. Object-Oriented Programming (ECOOP), 2018.

[4] S. Rahaman et al., "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Java Projects," arXiv:1902.02343, 2019.

[5] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?" in Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE), 2016.

[6] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. Yao, and N. Meng, "Automatic Detection of Java Cryptographic API Misuses: Are We There Yet?" IEEE Trans. Softw. Eng., 2022.

[7] SpotBugs, “SpotBugs: Find bugs in Java programs,” 2025. [Online]. Available: https://spotbugs.github.io/

[8] Find Security Bugs, “Find Security Bugs: SpotBugs plugin for security audits,” 2025. [Online]. Available: https://find-sec-bugs.github.io/

[9] OWASP Foundation, “OWASP Top 10:2021—A02:2021 Cryptographic Failures,” 2021. [Online]. Available: https://owasp.org/Top10/A02_2021-Cryptographic_Failures/

[10] OWASP Foundation, “OWASP Cryptographic Storage Cheat Sheet,” 2025. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

[11] Oracle, “SecureRandom (Java Platform SE documentation),” 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/security/SecureRandom.html

[12] Oracle, “Java Cryptography Architecture Standard Algorithm Name Documentation,” 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/17/docs/specs/security/standard-names.html

[13] Oracle, “Java Cryptography Architecture (JCA) Reference Guide,” 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/17/security/java-cryptography-architecture-jca-reference-guide.html

[14] National Institute of Standards and Technology (NIST), “Transitioning the Use of Cryptographic Algorithms and Key Lengths,” NIST SP 800-131A Rev. 2, 2019.

[15] National Institute of Standards and Technology (NIST), “Recommendation for Password-Based Key Derivation,” NIST SP 800-132, 2010.

[16] MITRE, “CWE-321: Use of Hard-coded Cryptographic Key,” 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/321.html

[17] MITRE, “CWE-327: Use of a Broken or Risky Cryptographic Algorithm,” 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/327.html

[18] MITRE, “CWE-295: Improper Certificate Validation,” 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/295.html

[19] E. Rescorla, “HTTP Over TLS,” RFC 2818, IETF, 2000.

[20] S. Ma et al., “CDRep: Automatic Repair of Cryptographic Misuses in Android Applications,” in Proc. ACM Asia Conf. Computer and Communications Security (AsiaCCS), 2016.

[21] L. Singleton, R. Zhao, M. Song, and H. P. Siy, “CryptoTutor: Teaching Secure Coding Practices through Misuse Pattern Detection,” in Proc. ACM Conf. on Information Technology Education (SIGITE), 2020.

[22] M. Kechagia, S. Mechtaev, F. Sarro, and M. Harman, “Evaluating Automatic Program Repair Capabilities to Repair API Misuses,” IEEE Trans. Softw. Eng., 2021.

[23] Oracle, “Java Secure Socket Extension (JSSE) Reference Guide,” 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/17/security/java-secure-socket-extension-jsse-reference-guide.html