Cryptographic API Misuses in Industry: A Case Study on Prevalence and Remediation

Joilton Almeida de Jesus[1], **Luís Henrique Vieira Amaral**[1], **Rodrigo Bonifácio**[1]

[1] Departamento de Ciência da Computação
Universidade de Brasília (UnB)
Brasília – DF – Brazil

{joilton.jesus, luis.amaral}@aluno.unb.br, rbonifacio@unb.br

Abstract. *Correct use of cryptographic APIs is crucial for data security in enterprise systems, yet remains challenging. This paper reports our experience applying CogniCrypt and CryptoGuard to detect cryptographic API misuses in 17 Java artifacts from a large agricultural research company. We identified 67 vulnerabilities—primarily involving insecure modes (e.g., AES/ECB) and key management issues—with 34 concentrated in a shared architectural component. Through a developer focus group and manual remediation, we assessed the tools' effectiveness and developers' perceptions, highlighting challenges in Static Application Security Testing (SAST) adoption and legacy code maintenance. As a practical contribution, we share our experience fixing the vulnerabilities and outline a migration strategy—necessary to ensure continued system functionality—that supports algorithm coexistence during and API compatibility.*

1. Introduction

The growing landscape of digital threats has elevated software security to a central concern for organizations. Encryption plays a crucial role in safeguarding sensitive data; however, its effectiveness relies heavily on the correct use of cryptographic Application Programming Interfaces (APIs). Improper usage of crypto APIs—often stemming from the complexity of these APIs and insufficient documentation—can introduce critical vulnerabilities, even in systems that are otherwise considered secure [Nadi et al. 2016, Acar et al. 2016]. The importance of the correct usage of cryptographic APIs is also highlighted by industry standards, such as the OWASP API Security Top 10, which lists as major risks problems like broken object-level authorization and improper assets management, areas where incorrect use of cryptography is often a contributing factor [OWASP Foundation 2023].

The significance of this issue extends beyond APIs to general web applications, as evidenced by the cornerstone OWASP Top 10 for Web Application Security Risks. In its latest iteration, "Cryptographic Failures" (A02:2021) is ranked as the second most critical risk, encompassing issues from the failure to encrypt data in transit or at rest to the use of weak cryptographic algorithms and poor key management. This high-ranking position underscores that the very types of API misuses analyzed in this paper are a direct manifestation of one of the most severe threats facing software systems today. [OWASP Foundation 2021]

Static Application Security Testing (SAST) tools support the detection of cryptographic API misuses [Hazhirpasand et al. 2020, Trautsch et al. 2023]. Nonetheless, their adoption is hindered by several challenges, including difficulties in integrating these tools into existing development workflows, the high incidence of false positives, and the requirement for specialized skills to accurately interpret the results [Ami et al. 2022]. Moreover, there is a scarcity of empirical evidence demonstrating how state-of-the-art static analysis tools—such as CogniCrypt [Krüger et al. 2017] and CryptoGuard [Rahaman et al. 2019]—can be effectively integrated into organizations that develop enterprise systems.

Seeking to fill this gap in the literature, we present a case study on detecting and fixing cryptographic API misuses in enterprise systems of an agricultural research company. The scope was intentionally defined to support a comprehensive and detailed analysis of a critical class of vulnerabilities. While other security issues like injection flaws are equally important, cryptographic errors can silently undermine the very foundation of data protection. Our research yielded three contributions. First, we found that a shared architectural component accounted for 34 of the 67 cryptographic API misuses identified across 17 enterprise systems in the company where we conducted our case study. Second, collaboration with the company's development teams revealed a culture that delegates security responsibilities to infrastructure, while highlighting the educational role of SAST tools in raising developer security awareness. Finally, we show that all 67 vulnerabilities can be manually remediated and propose a migration strategy to manage legacy data encrypted under the previous, insecure standard, ensuring continued API compatibility.

Section 2 provides the necessary background on the complexities of cryptographic APIs and the static analysis tools used to detect their misuses in our context. Sections 3 and 4 describe our research methodology and present the key findings. Finally, Section 5 concludes the paper.

2. Background

Cryptographic APIs, such as the Java Cryptography Architecture (JCA), are designed to abstract the complexity of cryptographic algorithms, offering developers interfaces to implement security [Nadi et al. 2016, Meng et al. 2018]. However, the improper use of these APIs is a common source of vulnerabilities. Developers frequently face difficulties due to a lack of documentation, the existence of poor examples, and unclear guidelines for secure configuration [Nadi et al. 2016, Meng et al. 2018]. Common errors include the use of obsolete or weak algorithms, including the use of Data Encryption Standard (DES) or unsafe modes of operation such as Electronic Codebook (ECB), improper key management (e.g., keys embedded as constants in the source code, weak keys), and incorrect handling of initialization vectors (IVs). Nadi et al. [Nadi et al. 2016] emphasize that the difficulty in understanding the correct sequence of method calls and necessary parameters contributes to insecure implementations.

For instance, Listing 1 shows an example of a misuse of the Java Cryptography Architecture (JCA) API. The issue resides in how the cipher is instantiated and initialized, particularly when the Electronic Code Book (ECB) mode is used (see Line 1 of Listing 1). The ECB mode is insecure because identical clear data blocks result in identical encrypted blocks, allowing an attack via pattern analysis [Dworkin 2001]. Figure 1 highlights the

fragility of the ECB operating mode. The original image is visible on the left, and the center image is encrypted using the ECB mode. One can notice some patterns that might reveal the outlines of the original image. In contrast, the image on the right shows the result of encrypting the original image using the Counter (CTR) mode. In the right image, it is not possible to identify any pattern that could reveal the content and allow an attacker to obtain the encrypted data.

```
1 Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
2 final SecretKeySpec secretKey = new SecretKeySpec(key, "AES");
3 cipher.init(Cipher.ENCRYPT_MODE, secretKey);
```

Listing 1. Improper cryptography API usage detected.

Figure 1. Image encrypted with ECB mode. (Source: Adapted from [Artiles et al. 2019])

To mitigate the risks associated with the improper use of cryptographic APIs, static code analysis tools have been developed. They inspect the source code or bytecode without executing the programs, searching for patterns of insecure usage. Two state-of-the-art crypto API misuse detectors for JCA are CogniCrypt [Krüger et al. 2017] and CryptoGuard [Rahaman et al. 2019]. CogniCrypt is a tool designed to assist developers in the proper use of encryption APIs in Java. It utilizes the *Cryptographic Specification Language* (CrySL) [Krüger et al. 2021] to define formal rules for safe usage. CogniCrypt can operate as an IDE plugin, providing real-time feedback, or via the command line for the analysis of existing projects. Its components include CogniCryptSAST for analysis and CogniCryptGen [Krüger et al. 2020] for generating secure code. CryptoGuard is another static analysis tool for Java, focused on detecting cryptographic vulnerabilities with high precision. It employs context-sensitive interprocedural data flow analysis to identify issues such as the use of weak keys, insecure algorithms, and the reuse of IVs. According to Rahaman et al. [Rahaman et al. 2019], CryptoGuard is designed to analyze large codebases and provide detailed reports.

Previous research has explored and compared both tools using open-source Java systems and synthetic benchmarks [Zhang et al. 2022, Ami et al. 2022, Torres et al. 2023, Firouzi et al. 2024]. However, qualitative assessments of these tools in industrial settings remain scarce, particularly within the context of Java enterprise systems. This work presents an experience report on using CogniCrypt and CryptoGuard to identify cryptographic API misuses in an industrial setting, capture development teams'

perceptions of the reported warnings, and assess the effort required to manually remediate the identified vulnerabilities.

3. Research Method

We conducted a case study at a large agricultural research institution using a structured, multi-stage methodology. This approach was designed to ensure the rigor of our findings and to offer a replicable framework for similar analyses in other industrial contexts. We conduct our research in four main stages.

(a) Artifact Selection

The first stage involved identifying the target systems for analysis. In collaboration with the institution's development and maintenance teams, we selected a representative set of 17 Java artifacts. This set comprised eight enterprise systems, seven web services that provide interfaces to these systems, and two architectural components containing reusable classes. These artifacts were chosen because they are considered critical for the institution's operational continuity, represent diverse levels of implementation complexity, and because the shared architectural components could be a source for propagating vulnerabilities.

(b) SAST Execution and Result Analysis

We used two state-of-the-art static analysis tools, CogniCrypt and CryptoGuard, to analyze the bytecode of the selected artifacts. The choice of these tools was based on their documented high precision in detecting cryptographic API misuses in Java. We executed the tools via their command-line interfaces and collected the generated reports. The results from both tools were then consolidated and manually cross-verified to eliminate any potential false positives and to categorize the findings.

(c) Developer Focus Group

After identifying and categorizing the vulnerabilities, we conducted a focus group session with members of the development teams responsible for the analyzed systems. The goals of this session were threefold: (i) to present and validate the findings, (ii) to understand the root causes of the misuses from the developers' perspective, and (iii) to collaboratively discuss and prioritize a remediation plan. The discussion was semi-structured, guided by questions about their awareness of secure coding practices, their previous experiences with SAST tools, and the perceived challenges in maintaining legacy code.

(d) Remediation and Validation

Based on the priorities defined in the focus group, the final stage involved the manual remediation of the identified vulnerabilities. We implemented the necessary code changes, focusing first on the most critical and widely propagated issues found in the shared architectural components. After applying the fixes, we re-ran CogniCrypt and CryptoGuard to validate that the vulnerabilities were successfully eliminated. Finally, we executed the projects' automated test suites to ensure that our changes did not introduce functional regressions. The following section details the findings and outcomes obtained by applying this methodology.

4. Results

This section presents the results obtained by applying the research method we described in the previous section.

4.1. Identification of Cryptographic API Misuses

The outcomes of CogniCrypt and CryptoGuard revealed a total of 67 occurrences of cryptographic API misuses in the selected Java artifacts. The vulnerabilities were found mainly in Component 1 (a core architectural library), which accounted for 34 of the identified misuses. Table 1 summarizes the main types of vulnerabilities (categorized by the Common Weakness Enumeration - CWE [MITRE Corporation 2024]) detected, the tools that identified them, and the number of occurrences for each.

Table 1. Vulnerabilities found in the static analyses.

Type of Misuse (Associated CWE)	Description	Tool(s)	Occurrences
Use of insecure algorithm (CWE-327)	Ex: AES/ECB, DES/ECB	CogniCrypt, CryptoGuard	9
Incorrect key generation (CWE-320)	Parameters not generated correctly	CogniCrypt	9
Incorrect preparation of key material (CWE-321)	Parameters not generated correctly	CogniCrypt	9
Use of outdated SSL protocol (CWE-326)	Use of 'SSL' instead of TLSv1.2+	CogniCrypt	5
Incorrect generation of TrustManagers (CWE-295)	Insecure configuration	CogniCrypt	3
Use of unreliable PRNG (CWE-338)	Ex: `java.util.Random` for cryptographic purposes	CryptoGuard	4
Use of constant key in the code (CWE-547)	Hardcoded keys	CryptoGuard	16
Use of unreliable HostNameVerifier (CWE-295)	Accepts all hostnames	CryptoGuard	3
Use of unreliable TrustManager (CWE-295)	Accepts all certificates	CryptoGuard	8
Use of insecure HTTP protocol (CWE-319)	HTTP instead of HTTPS	CryptoGuard	1

The most recurrent vulnerabilities include the use of an insecure cryptographic mode (i.e., `AES/ECB/PKCS5Padding`) (CWE-327), the generation of keys from constants in the code (CWE-547), and the use of insecure transport protocols such as SSL instead of TLSv1.2 or higher (CWE-326). Component 1, being a shared library, propagated several of these vulnerabilities to the systems that depended on it.

4.2. Developer Perceptions and Discussion

After identifying the cryptographic API misuses, we conducted a focus group session with the development teams to share the findings and decide on appropriate measures based on the results.

Regarding the origin of the misuses, the focus group revealed that, although developers consider security important, it is frequently neglected in favor of delivering functionality. Also, there is an excessive reliance on the IT infrastructure to ensure security, and a limited knowledge of secure coding practices and the risks associated with legacy architectural components. One developer stated: "*Our goal is to deliver what the client requested. [. . .] We believe that security should be delegated to the application server configuration, the VPN . . . , and ensured by the infrastructure only*.". This highlighted a cultural gap where security was not seen as a shared responsibility that includes software development.

The team was generally unaware of the specific risks associated with their cryptographic implementations, such as the weakness of the ECB mode. When presented with the findings, they understood the theoretical risks but had not previously considered them a priority. The discussion also highlighted practical challenges in addressing vulnerabilities, especially in legacy architectural components. The interdependence of these

elements and the considerable risk of a "snowball effect" were central concerns. For instance, one of the participants stated that *"[. . .] if you want to update the cryptography architectural component, you would also need to update other dependencies, including the Java version we use here. It's a snowball effect, where one problem leads to another, and before you realize it, you're dealing with a significantly outdated environment."*

Despite previous experiences with SAST tools not always being positive, the analysis of concrete vulnerabilities prompted a reassessment of their value. Developers emphasized the educational potential of these tools and expressed interest in integrating them more effectively into the development process.

> *"These tools can help us become more familiar with security issues, which are sometimes overlooked. These tools gradually reveal where errors occur, right? So, I believe it's good practice to reconsider our development process and add this security layer."*

The participants acknowledged the critical nature of the vulnerabilities, especially those in reusable components developed over a decade ago, and agreed to prioritize their remediation.

4.3. Fixing the Cryptographic Misuses

We carried out manual corrections of the critical vulnerabilities identified. Here we outline the process and the impact of the fixes using representative examples of the analyzed artifacts. Due to space limitations, we report only three patterns of fixes implemented in a widely used architectural component, where a significant number of failures were originally detected.

(Fix 1) Correction of CWE-327 (Use of Insecure Algorithm). The use of `AES/ECB/PKCS5Padding` (see Listing 1) was identified in encryption and decryption methods. The fix involved replacing the ECB mode with the *Counter* (CTR) mode, `AES/CTR/NoPadding`, and introducing a correct use of Initialization Vectors (IVs) randomly generated for each encryption operation. This decision aligns with CogniCrypt's recommendations, which suggest using secure operating modes such as CTR. Listing 2 shows the modified code snippet, highlighting the instantiation of the AES cipher in CTR mode and the explicit use of an Initialization Vector (`IvParameterSpec`), in line with recommended secure practices.

```
1 Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
2 IvParameterSpec ivSpec = new IvParameterSpec(iv);
3 final SecretKeySpec secretKey = new SecretKeySpec(key, "AES");
4 cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);
```

Listing 2. Corrected usage example with AES/CTR and IV.

(Fix 2) Correction of CWE-320 and CWE-547 (Incorrect Key Generation and Use of Constant Key). The encryption key was a constant `byte[]` in the source code, and the `SecretKeySpec` class was instantiated insecurely. To correct this, key derivation was implemented using PBKDF2WithHmacSHA256 (`PBEKeySpec`),

based on a password (obtained securely and not stored directly) and a random *salt* generated with `SecureRandom`. The java.security.SecureRandom class is the standard, cryptographically strong pseudo-random number generator (PRNG) in Java, as it gathers entropy from the underlying operating system, making it suitable for cryptographic tasks. The password stored in a `char[]` was zeroed out after use to minimize its exposure in memory. The IV was also generated randomly with `SecureRandom` for each encryption operation. The *salt* and the IV were then concatenated (Base64 encoded) to the encrypted text to allow decryption.

(Fix 3) Correction of CWE-326 (Outdated SSL Protocol). Methods responsible for configuring secure connections (e.g., `sslCheck()`) instantiated `SSLContext` with the string `''SSL''`, allowing the negotiation of outdated protocols such as SSLv3. The correction involved the explicit specification of `TLSv1.2`. The decision to specify TLSv1.2 instead of the more recent TLSv1.3 was dictated by a critical technical constraint: the institution's legacy systems are standardized on Java version 1.7.0_80. This decision to remain on this specific Java version stems from significant organizational factors, including licensing model changes introduced with later versions and the imperative to maintain compatibility with other interdependent legacy systems. As the 1.7.0_80 platform supports TLSv1.2 but not TLSv1.3, adopting the latter would be unfeasible without a major platform migration, exemplifying the "snowball effect" concern previously raised by the developers. This fix also included the use of a `KeyStore` of type "JKS", and the configuration of `TrustManagerFactory` and `KeyManagerFactory` with secure algorithms such as "SunX509", ensuring proper certificate validation and the use of robust cipher suites. In addition, permissive TrustManagers and HostnameVerifiers that accepted any certificate or hostname were adjusted to perform strict validations.

In total, fixes (Fix 1), (Fix 2), and (Fix 3) resolved 43 warnings, modifying 336 lines of code. Addressing the remaining 24 warnings required changes to 688 lines of code. We verified the effectiveness of these fixes by re-running the SAST tools to confirm that the modifications successfully eliminated the issues, as no further warnings were generated. Finally, we successfully executed the system build and functional tests, ensuring the stability and correctness of the updated code.

While the revised codebase demonstrates stability, essential updates to its cryptographic algorithms introduced incompatibility with pre-existing data structures. Consequently, a transitional strategy was adopted, involving the concurrent operation of both the legacy and revised code. This approach allows the legacy system to decrypt older data for subsequent re-encryption under the new cryptographic standard. Concurrently, safeguards were instituted to prohibit the legacy system from encrypting new data, thereby mandating the revised code for all current and future encryption operations. To minimize disruption to client applications, the method signatures of the cryptographic functions were maintained, ensuring continued API-level compatibility.

5. Conclusions

This study presented an experience report on the exploration of two state-of-the-art static cryptographic API misuse detectors—CogniCrypt and CryptoGuard—within the enter-

prise systems of a large Brazilian research organization.[1] We found that almost half of the 67 detected vulnerabilities originated from a reusable component that propagated these vulnerabilities across multiple systems. We also outlined the development teams' perceptions of the reported vulnerabilities and highlighted their decisions to integrate these tools into the company's pipelines and to address the detected vulnerabilities. The main takeaway messages from this practical experience are:

- The relevance of using static analysis tools goes beyond merely finding vulnerabilities, as they foster cultural change and promote knowledge acquisition, helping to disseminate awareness about vulnerabilities that might not be fully understood by development teams in organizations that develop internal enterprise systems.
- The feasibility of fixing a significant number of vulnerabilities (67 in our case) present in legacy systems, even through a manual approach. In our case, we fixed all 67 warnings detected by CogniCrypt and CryptoGuard by modifying 1024 lines of code in 12 Java classes.

As future work, we aim to reproduce the manual fixes using foundation models (i.e., small and large language models) to evaluate whether the fixes can be automated. We also plan to extend this study to other systems within the organization to corroborate our findings. Depending on the outcomes, we may extend this approach to other enterprise systems within our organization.

References

Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M. L., and Stransky, C. (2016). You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305.

Ami, A. S., Cooper, N., Kafle, K., Moran, K., Poshyvanyk, D., and Nadkarni, A. (2022). Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 614–631. IEEE.

Artiles, J., Chaves, D., and Pimentel, C. (2019). Image encryption using block cipher and chaotic sequences. *Signal Processing: Image Communication*, 79.

Dworkin, M. (2001). Recommendation for block cipher modes of operation. Technical report, NIST Special Publication 800-38A. Disponível em: `https://csrc.nist.gov/publications/detail/sp/800-38a/final`.

Firouzi, E., Ghafari, M., and Ebrahimi, M. (2024). Chatgpt's potential in cryptography misuse detection: A comparative analysis with static analysis tools. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 582–588.

Hazhirpasand, M., Ghafari, M., and Nierstrasz, O. (2020). Java cryptography uses in the wild. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6.

[1]Due to the sensitive nature of the findings, which involve security vulnerabilities in a partner institution's corporate systems, the detailed code and specific fixes cannot be shared publicly. This data is managed securely in the institution's internal configuration and version control systems.

Krüger, S., Ali, K., and Bodden, E. (2020). Cognicryptgen: generating code for the secure usage of crypto apis. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20)*, pages 185–198, New York, NY, USA. Association for Computing Machinery.

Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., Göpfert, F., Günther, F., Weinert, C., Demmler, D., and Kamath, R. (2017). Cognicrypt: supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*, pages 931–936. IEEE Press.

Krüger, S., Späth, J., Ali, K., Bodden, E., and Mezini, M. (2021). CrySL: An extensible approach to validating the correct usage of cryptographic APIs. *IEEE Transactions on Software Engineering*, 47(11):2382–2400.

Meng, N., Nagy, S., Yao, D. D., Zhuang, W., and Argoty, G. A. (2018). Secure coding practices in java: challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 372–383, New York, NY, USA. Association for Computing Machinery.

MITRE Corporation (2024). CWE – Common Weakness Enumeration. `https://cwe.mitre.org/`. Acesso em: 20 mai. 2025.

Nadi, S., Krüger, S., Mezini, M., and Bodden, E. (2016). Jumping through hoops: why do Java developers struggle with cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, pages 935–946, New York, NY, USA. Association for Computing Machinery.

OWASP Foundation (2021). OWASP Top 10:2021 - The Ten Most Critical Web Application Security Risks. `https://owasp.org/www-project-top-ten/`. Acesso em: 20 jul. 2025.

OWASP Foundation (2023). OWASP API Security Top 10. `https://owasp.org/API-Security/editions/2023/en/0x11-t10/`. Acesso em: 13 jul. 2025].

Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., and Yao, D. D. (2019). CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*, pages 2455–2472, New York, NY, USA. Association for Computing Machinery.

Torres, A., Costa, P., Amaral, L., Pastro, J., Bonifácio, R., d'Amorim, M., Legunsen, O., Bodden, E., and Dias Canedo, E. (2023). Runtime verification of crypto apis: An empirical study. *IEEE Transactions on Software Engineering*, 49(10):4510–4525.

Trautsch, A., Herbold, S., and Grabowski, J. (2023). Are automated static analysis tools worth it? an investigation into relative warning density and external software quality on the example of apache open source projects. *Empirical Software Engineering*, 28(3):66.

Zhang, Y., Kabir, M. M. A., Xiao, Y., Yao, D. D., and Meng, N. (2022). Automatic detection of java cryptographic api misuses: Are we there yet. *IEEE Transactions on Software Engineering*, page 1–1.