The Journal of Systems and Software 216 (2024) 112135

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Time to separate from StackOverflow and match with ChatGPT for encryption[☆]

Ehsan Firouzi, Mohammad Ghafari [*]

Technische Univertität Clausthal, Germany

ARTICLE INFO

Keywords:
Java cryptography
Symmetric encryption
Security
StackOverflow
ChatGPT

ABSTRACT

Cryptography is known as a challenging topic for developers. We studied StackOverflow posts to identify the problems that developers encounter when using Java Cryptography Architecture (JCA) for symmetric encryption. We investigated security risks that are disseminated in these posts, and we examined whether ChatGPT helps avoid cryptography issues. We found that developers frequently struggle with key and IV generations, as well as padding. Security is a top concern among developers, but security issues are pervasive in code snippets. ChatGPT can effectively aid developers when they engage with it properly. Nevertheless, it does not substitute human expertise, and developers should remain alert.

1. Introduction

The analysis of security issue reports across numerous open-source projects on GitHub revealed a concerning trend: the proliferation of security issues is on the rise, while their resolution progresses slowly and only a small group of developers are involved in the process (Bühlmann and Ghafari, 2022). Despite the crucial role of Cryptography in the seamless integration of security into our digital world, developers struggle with existing cryptography libraries. These libraries often do not support common operations, lack sufficient abstraction, and have poor documentation quality (Mindermann et al., 2018; Hazhirpasand et al., 2021a; Patnaik et al., 2019). Hence, API misuses are likely and so does the presence of security vulnerabilities. For instance, the analysis of cryptography in 489 open-source Java projects has revealed that 85% included API misuses (Hazhirpasand et al., 2020). These issues are also present in proprietary software systems. Notably, researchers have identified weak encryption algorithms and legacy encryption modes in critical infrastructure (Wetzels et al., 2023).

Java Cryptography Architecture (JCA) is the most widely adopted cryptography API, and symmetric encryption is the foremost adopted cryptography operation in software systems. Most of the top 100 cryptography questions on StackOverflow, sorted by views and score, are about symmetric encryption. Similarly, it is adopted in 64% of the top 100 GitHub projects, sorted by stars, that use JCA (Nadi et al., 2016).

Unlike prior studies that are broad, in this paper, we focused specifically on symmetric encryption with JCA, providing a detailed view on its challenges for developers. We blended qualitative and quantitative analyses to uncover developer issues and common security risks observed on the StackOverflow website, and we evaluated the effectiveness of ChatGPT in addressing these issues. In particular, we investigated the following three research questions:

RQ$_1$: What are common developer challenges in symmetric encryption?

We manually inspected 400 StackOverflow posts that are about JCA symmetric encryption and found that the majority of reported problems (*i.e.,* 214 posts) are at the "cipher object initialization" stage. The primary challenges were in key and initialization vector (IV) management. There were 116 posts that highlighted issues encountered during the "cipher object instantiation" stage, particularly related to padding, encryption mode, and algorithm selection. We identified 99 posts that reported problems during the "encryption/decryption" stages, with encoding being the most prevalent issue. The "transmission of parameters (key/IV)" had the lowest number of reported issues (*i.e.,* 21 posts). They were mainly linked to encrypting keys using various algorithms and facing difficulties related to keystores.

We also found a total of 128 exceptions, with "BadPaddingException" and "InvalidKeyException" is the most common, occurring 106 times in the posts. Nevertheless, only 45 exceptions (*i.e.,* 42%) were directly related to padding and key issues, making it challenging for developers to pinpoint the root causes.

RQ$_2$: What are the security risks present in the shared JCA code on StackOverflow?

[☆] Editor: Raffaela Mirandola.

[*] Corresponding author.
E-mail addresses: ehsan.firouzi@tu-clausthal.de (E. Firouzi), mohammad.ghafari@tu-clausthal.de (M. Ghafari).

https://doi.org/10.1016/j.jss.2024.112135
Received 29 January 2024; Received in revised form 16 May 2024; Accepted 10 June 2024
Available online 12 June 2024

Table 1
Symmetric algorithms in JCA.

Symmetric Encryption	Algorithms
Block Cipher	AES DES DESede(3DES) Blowfish
Stream Cipher	RC2, RC4, RC6 ChaCha20

The examination of 13 security rules in 400 posts revealed a striking number of 327 posts (*i.e.,* 82%) with security violations. The use of ECB and CBC encryption modes were the most common violations followed by hard-coded keys. We collected the symptoms of security violations and searched for them across all 3426 symmetric encryption posts on StackOverflow. The findings revealed 5305 violations in 3174 posts, averaging 1.7 violations in 92% of StackOverflow posts. In general, we observed a lack of adherence to best practices, such as those for password-based key generation, but the adoption of 256-bit keys for AES and GCM encryption mode have increased in recent years.

RQ$_3$: How effective is ChatGPT for addressing developer issues in symmetric encryption?

We provided 100 StackOverflow questions to ChatGPT (GPT-3.5) and recorded the answers. We observed that it provided a "working" solution. Nevertheless, they are problematic from a security perspective. Precisely, when we provided the exact StackOverflow questions to ChatGPT, it transferred almost every violation from the question to its answer. When we explicitly prompted to provide a "secure solution", it cleared violations in 42 questions. ChatGPT detected DES as an insecure algorithm and ECB as a vulnerable encryption mode almost in every case. We could clear violations in 26 more questions when we pinpointed the exact line where a violation existed. In the end, it adopted a weak key generation function in two questions and included a CBC mode in 30 questions, which ChatGPT did not flag as a violation even for client–server scenarios.

In summary, the contributions of this paper are the following.

- We provided a comprehensive study on Java developer challenges in symmetric encryption.
- We discovered that security violations are prevalent in the code examples shared on the StackOverflow website, threatening novices who blindly copy and paste code into their programs.
- We highlighted the potential of ChatGPT in complementing human expertise and demonstrated its superiority in evaluating code security on a line-by-line basis.
- We shared our dataset of symmetric encryption posts, along with the detailed results of our manual investigations, the security expression rules, as well as the links to ChatGPT responses.[1]

The rest of this article is organized as follows. We describe our research methodology in Section 2. We present our results in Section 3 and discuss our findings in Section 4. We explain threats to the validity of our study in Section 5 and make an overview of related work in Section 6. Finally, we conclude this paper in Section 7.

2. Study setup

We relied on the Stack Exchange Data Dump released on March 8, 2023 (Stackexchange, 2023). We extracted the StackOverflow posts that are related to symmetric encryption and randomly selected a representative subset of these posts for manual inspection.

[1] https://github.com/Ehsan-Firouzi/SymmetricEncryptionChallenges

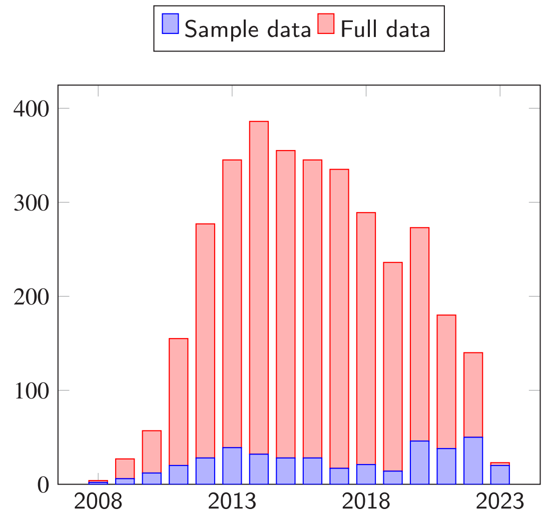


Fig. 1. Distribution of symmetric encryption posts.

2.1. Data collection

The `Cipher` class in JCA offers encryption and decryption functionalities. It supports both block and stream cipher symmetric encryption algorithms. In short, stream ciphers convert one symbol of plaintext directly into a symbol of ciphertext, whereas block ciphers, which are more modern, encrypt a group of plaintext symbols as one block. Table 1 lists these algorithms.

Full dataset. We searched the StackOverflow for cipher instances instantiated with a symmetric encryption algorithm. We utilized the regular expression (RegEx) shown in Listing 1 to search within both questions and accepted answers. We discarded posts where the patterns were only found within other answers, as viewers of the posts pay greater attention to the aforementioned sections. This filtering process resulted in 3426 posts.

```
Cipher\.getInstance\(("|\&quot)(AES|DES|DESede|RC|Blowfish|ChaCha20)
```

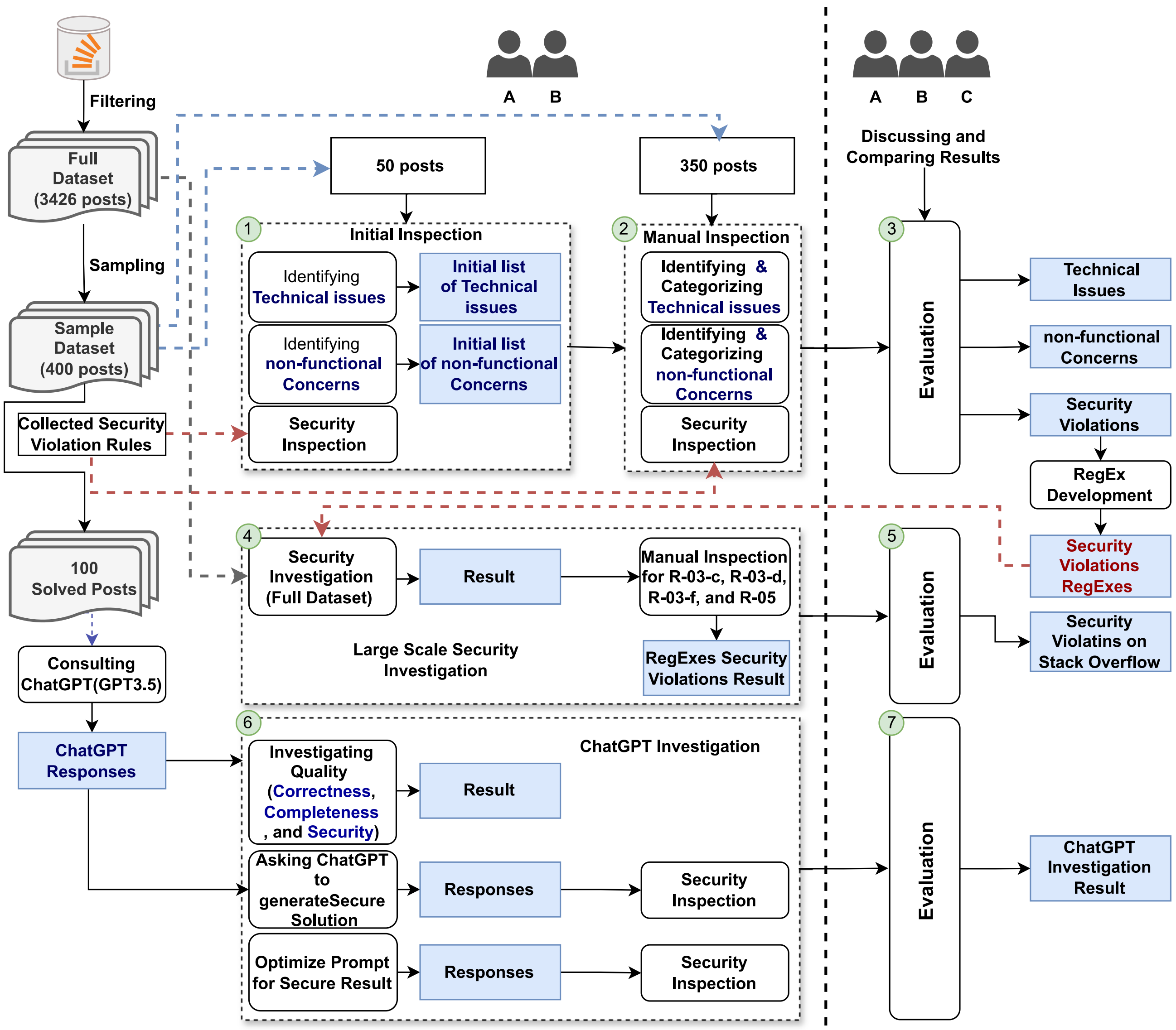
Listing 1: RegEx for capturing symmetric encryption uses in JCA

Sample dataset. We chose a subset of posts for a manual inspection, and to ensure that the findings represent the entire dataset, we included 400 posts. This sample size yields a 95% confidence level with a less than 5% (4.5%) margin of error. We sought to incline our sample data towards recent posts and those with high scores. Therefore, we selected 40% of the posts between 2020 and 2023, representing the most recent posts, 30% from the most popular ones (highest scores), and the other 30% completely random.

In summary, we gathered a total of 3426 posts for our "full dataset" and chose 400 posts to form our "sample dataset". Fig. 1 illustrates the distribution of these posts in each year, and Table 2 lists further details about our data.

Table 2
Further details about the collected StackOverflow posts.

Dataset	#Solved Posts	#Pending Posts	Score AVG.	View AVG.
Full	1,723	1,703	2	3,263
Sample	251	149	5	13,892



Filtering
Full Dataset (3426 posts)
Sampling
Sample Dataset (400 posts)
Collected Security Violation Rules
100 Solved Posts
Consulting ChatGPT(GPT3.5)
ChatGPT Responses
A B
50 posts
350 posts
1 Initial Inspection
Identifying Technical issues
Initial list of Technical issues
Identifying non-functional Concerns
Initial list of non-functional Concerns
Security Inspection
2 Manual Inspection
Identifying & Categorizing Technical issues
Identifying & Categorizing non-functional Concerns
Security Inspection
A B C
Discussing and Comparing Results
3 Evaluation
Technical Issues
non-functional Concerns
Security Violations
RegEx Development
Security Violations RegExes
4 Security Investigation (Full Dataset)
Result
Manual Inspection for R-03-c, R-03-d, R-03-f, and R-05
Large Scale Security Investigation
RegExes Security Violations Result
5 Evaluation
Security Violatins on Stack Overflow
6 ChatGPT Investigation
Investigating Quality (Correctness, Completeness, and Security)
Result
Asking ChatGPT to generateSecure Solution
Responses
Security Inspection
Optimize Prompt for Secure Result
Responses
Security Inspection
7 Evaluation
ChatGPT Investigation Result

Fig. 2. Overview of our research methodology.

2.2. Methodology

Fig. 2 shows the overview of our methodology. Three individuals (A1, A2, and A3) were involved, each possessing practical knowledge in cryptography and over three years of Java programming experience.

2.2.1. Developer challenges

To identify and categorize the technical issues (*i.e.,* root causes of the problems), and identify developers' non-functional concerns, we conducted a lightweight open-coding-like process. This process involved three phases as follows:

Phase I (Collecting an Initial List of Technical Issues and Concerns). A1 and A2 reviewed 50 posts (questions, their accepted answers, and comments) together. For each post, guided by official sources (Oracle, 2021; Ferguson et al., 2011; Chung et al., 2012), they identified and recorded the technical issues (i.e., root causes of the problems), the stage at which these issues occurred, and non-functional concerns that were explicitly raised. This step resulted in the compilation of two initial lists, including (1) technical issues and (2) non-functional concerns.

Phase II (Categorization). A1 and A2 independently reviewed the remaining 350 posts to complete the identification and categorization of technical issues and non-functional concerns.

Phase III (Evaluation). A1 and A2 compared their findings and categorizations. They discussed any disagreements until a consensus was reached. In instances where differing opinions persisted for certain posts, A3, who had not yet reviewed those posts, was consulted. Ultimately, the results were finalized using a majority voting mechanism (Cohen's k = 0.86).

Table 3
Security violation rules applicable to symmetric encryption.

Stage	Rule-ID	CWE-ID	Violation	JCA API
Cipher Instantiation	R-01	CWE-327	Using weak Algorithm	Cipher
	R-02-a	CWE-327	Using ECB encryption mode	Cipher
	R-02-b	CWE-327	Using CBC encryption mode	Cipher
Cipher Initialization-Key	R-03-a	CWE-798	Using static or constant key	SecretKeyspec
	R-03-b	CWE-330	Using static salt for key derivation	PBEKeySpec PBEParameterSpec
	R-03-c	CWE-326 CWE-330	Using salt<64 bits for key derivation.	PBEKeySpec PBEParameterSpec
	R-03-d	CWE-326 CWE-330	Using iterations<1000 for key derivation	PBEKeySpec PBEParameterSpec
	R-03-e	CWE-259	Using hard-coded password	PBEKeySpec
	R-03-f	CWE-330	Using weak Random function for generating secret key	KeyGenerator
	R-03-g	CWE-327	Using weak algorithms for generating secret key	SecretKeyFactory
Cipher Initialization-IV	R-04-a	CWE-330	Using static IV	IvParameterSpec
	R-04-b	CWE-330	Using a “badly-derived” Initialization Vector (IV)	IvParameterSpec
Parameters Transmission	R-05	CWE-798	Loading a keystore using an input stream with a constant and non-null password value	KeyStore

2.2.2. *Security analysis*

We followed three phases to uncover security risks. We conducted them simultaneously during the investigation of developer challenges.

Phase I (Collecting Security Violation Rules). Recognizing the challenge posed by the incomplete nature of code snippets on StackOverflow, making it difficult to assess their security using available tools, we opted for a meticulous manual approach. We consulted the state-of-the-art in crypto misuse analysis and compiled a list of security violation rules for symmetric encryption. These rules, detailed in Table 3, were primarily sourced from tools such as CryLogger (Piccolboni et al., 2021) and CogniCrypt (Krüger et al., 2019), as well as insights from a recent study in this domain (Zhang et al., 2023).

Phase II (Security Violations Investigation). A1 and A2 separately examined the security of each post according to rules collected in Phase I. In particular, for each post in the sample dataset, they checked whether any of the violation rules were present in the question or accepted answer, and collected the code snippets associated with every violation.

Phase III (Evaluation). A1 and A2 compared their results. They discussed any disagreements until a consensus was reached. In instances where differing opinions persisted for certain posts, A3 was consulted. Ultimately, the results were finalized using a majority voting mechanism (Cohen’s k = 0.86). At the end of this phase, through the study of insecure code snippets, we developed a set of regular expressions to enable us to detect security violations among code snippets. Recognizing the challenge of automatically resolving values passed as parameters in functions, often overlooked by static analysis tools (Afrose et al., 2023), we sought ways to enhance precision. To address this, we manualized a part of our investigation, specifically for rules R-03-c, R-03-d, R-03-f, and R-05, and we considered this concern in the regular expressions (RegExes).

Phase IV (Large Scale Security Analysis). To acquire overall insights into the state of security in the StackOverflow posts:

1. We applied the regular expressions extracted in the previous phase to the full dataset (3426 posts) for a comprehensive security analysis of all StackOverflow posts.
2. Following this, A1 and A2 separately conducted manual checks on the results of the above-mentioned rules (in Section Phase III of Section 2.2.2) to identify insecure code snippets that violated these specified rules.
3. Similar to phase III in Section 2.2.1 A1, A2, and A3 evaluated the results (Cohen’s k = 0.91). The code snippets identified by RegExes for rules R-01, R-02-a, R-02-b, and R-03-g do not require additional manual checking, as they are confirmed to be 100% vulnerable.

2.2.3. *ChatGPT analysis*

We consulted ChatGPT (GPT-3.5) to evaluate its potential in assisting developers with cryptography-related queries. We randomly selected 100 posts with accepted answers from our pool of 400 sample posts (Evaluating the correctness and completeness of generated answers for questions without accepted answers is challenging because there is no answer to compare with the generated answers. This makes the study challenging and introduces a higher risk of human bias). Each chosen post *included at least one security violation.*

1. Firstly, we presented each “exact” question from the posts to ChatGPT, followed by the command “answer this StackOverflow question and provide a code example”.’ We collected its responses.
2. A1 and A2 independently and manually:
   2.1. Examined the ChatGPT answers from a security perspective (by the help of collected security violation rules in Table 3)
   2.2. They evaluated the correctness and completeness of the generated answers. In this regard, for each question they compared the ChatGPT answer with the accepted answer and assessed if the ChatGPT answer was correct or misleading (Correctness) and if the answer fully addressed the question(Completeness). as there is no accepted answer to compare with, making the study more challenging and with a higher risk of human bias.
3. like phase III in Section 2.2.1, A1 and A2 compared results, resolving disagreements for consensus. A3 was consulted for persistent differences. Final results were determined by majority voting (Cohen’s k = 0.82).
4. Secondly, we explicitly instructed ChatGPT to generate a “secure” code example. and checked the answers from a security perspective.
5. Lastly, for posts where ChatGPT’s response remained insecure, we searched how we can optimize the prompt to get a more secure answer.

3. Result

We present our findings about developer challenges. Then, we report our security investigation result, and lastly, we show to what extent ChatGPT can clear these issues.

3.1. *Developer challenges*

We present technical issues as well as developer concerns discussed in 400 sample posts.

Table 4
Distribution of technical issues per stage.

	Problem	#Issues	#Posts
Instantiation	Padding	89	116
	Encryption mode	49	
	EncryptionAlgorithm	15	
Initialization	Key	121	214
	IV	106	
Encryption/ Decryption	Encoding	54	99
	Update/doFinal	18	
Transmission	Key wrapping/Exchange	12	21
	Keystore	5	

3.1.1. *Technical issues*

We found a total of 450 technical issues in 400 posts. We categorized these issues into nine categories across four steps. The top common issues were related to the initialization step followed by issues in the instantiation step. Table 4 lists the most common issues in each encryption stage. We present these issues and share common root causes that we observed.

Cipher Instantiation. This stage is for creating a Cipher object in Java for encryption and decryption operations using a factory method. We discovered a total of 116 posts that were related to cipher instantiation. The most common root causes were padding, encryption mode, and encryption algorithm. In particular, 89 issues were associated with padding problems such as choosing an improper padding mode for a specific scenario, using different padding modes for encryption and decryption, and not explicitly specifying the padding mode. We identified 49 posts that had issues related to encryption mode. These issues included the absence of an explicitly specified cipher mode, problems arising from dependencies between encryption mode and initialization vector (IV), and choosing different encryption modes for encryption and decryption. Finally, we found 15 general posts about encryption algorithms such as how to adopt a specific algorithm.

Cipher Initialization. The Cipher object has to be initialized with the appropriate encryption mode, key, and any necessary parameters such as initialization vectors. Developers struggle the most in the cipher initialization step. Key and vector initialization were the two most common root causes in 214 posts. In particular, 121 posts were related to key initialization problems such as invalid key sizes (length), password-based key derivation problems, and using different keys for encryption and decryption. There were 106 posts that dealt with issues related to IV. We observed common issues such as how to generate IVs, the dependency of IVs on encryption modes, using different IVs for encryption and decryption, and considerations regarding the size of IVs.

Encryption/Decryption. The actual process of encryption or decryption occurs in this step, utilizing methods such as update and doFinal, along with encoding techniques. We found that one-fourth of the posts discuss issues related to the encryption/decryption phase. There were 54 posts about encoding issues, especially due to the use of different encoding systems during encryption and decryption. In 18 posts, developers could not determine when to use `update()` and `doFinal()` methods.[2]

Transmission. We require the encryption parameters, such as the encryption key, to decrypt a ciphertext. The number of posts that discussed transmission issues was the lowest (*i.e.,* 21 posts). The common issues that we observed were related to key wrapping and key exchange problems (*i.e.,* utilizing RSA and DH algorithms for exchanging symmetric keys) in 12 posts, and keystore-related issues in five posts.

[2] The former method is to process data in chunks, whereas the latter method is used for the final step of encryption or decryption operations.

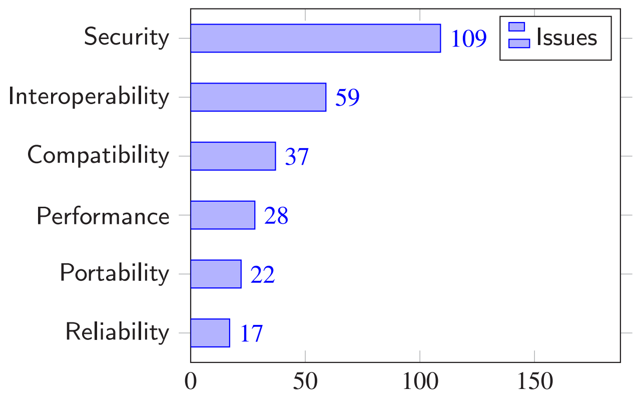


Fig. 3. Distribution of non-functional issues.

3.1.2. *Non-functional concerns*

We identified 272 non-functional concerns in 225 (out of 400) posts and categorized them into six main categories listed below. The most recurring concerns were about security and interoperability. Fig. 3 shows a breakdown of the result.

- Security: We assigned posts to this category if the user explicitly inquired about security or relevant hints were present in the discussions.
- Performance: Posts in this category were concerned about enhancing the efficiency of the code. These posts mainly discussed issues related to optimizing memory usage, managing resources effectively, and improving the execution speed of the code (aiming for faster and more responsive performance).
- Reliability: This category of posts aimed to develop code that is stable and reliable, minimizing crashes (Jalote et al., 2004) and ensuring proper thread management.
- Compatibility: These posts deal with compatibility-related challenges such as issues arising from the code's incompatibility with specific providers or Java versions.
- Portability: Posts that discuss the ability of the code to be seamlessly transferred across different environments or platforms.
- Interoperability: The posts within this category aimed to facilitate effective interaction and functionality between the code and code developed in other programming languages.

Security. The foremost concern, discussed in 109 posts, was about security. The primary concern, highlighted in 46 posts, revolved around ensuring the unpredictability of initialization vectors (IV). The second most prevalent concern pertained to the safety of encryption modes, mentioned in 32 posts. We identified 29 posts expressing worries about the secure derivation of encryption keys from passwords. Furthermore, 6 posts raised doubts about the security of chosen encryption algorithms.

Interoperability. We encountered 59 posts in which developers expressed concerns about interoperability. There were 9 posts addressing issues with PHP, 9 posts related to Node.js, 8 posts highlighted challenges with C#, and other posts (*i.e.,* 33) were related to other programming languages. We observed that the underlying causes of interoperability issues were primarily related to differences in default encoding and padding methods.

Compatibility. Developers discussed compatibility in 37 posts. The most common issues (*i.e.,* 14) were related to the JDK version. We identified 5 issues about the absence of native support for PKCS7 padding, 4 issues due to Android versions, and 3 related to different service providers.

Performance. We found 28 posts that included performance concerns, of which 11 posts were about memory concerns, and 19 posts discussed speed concerns. Memory matters were about OutOfMemoryException for large files, which needed the use of CipherInputStream and CipherOutputStream instead of `cipher.update()` and `cipher.doFinal()`. Speed matters were due to the buffer size or a lack of `CipherInputStream` and `CipherOutputStream`.

Portability. Developers were concerned about the portability of their code in 22 posts. The discussions unanimously revolved around the default values of parameters on providers when developers do not explicitly specify them.

Reliability. We found 17 posts that included reliability concerns, mainly related to unexpected behavior or crashes. The most common issue (discussed in 5 posts) was about Cipher not being thread safe.

Developers struggle a lot with key and IV generations, and padding. Security and interoperability are their top concerns.

3.2. Security risks

We investigated the presence of security issues in our sample posts. We then developed a set of regular expressions that helped us to find such issues in the full dataset.

3.2.1. Security rules

We found a total of 13 security rules in the literature that concern symmetric encryption. Table 3 lists these rules, indicates what exact APIs are involved, and provides CWE links. The items highlighted in orange text do not necessarily indicate the insecure pattern; instead, they represent bad practices that are not secure enough.

R-01. DES, RC2, and RC4 are outdated algorithms and have been compromised by modern cryptanalysis methods. While 3DES is not broken but it was deprecated by NIST (Barker and Mouha, 2017). Blowfish is generally regarded as a strong and secure algorithm, but it has a 64-bit block size, making this algorithm susceptible to birthday attacks, especially in HTTPS context (Bhargavan and Leurent, 2016). Moreover, the GnuPG project recommended against using Blowfish for encrypting files larger than 4 GB due to its small block size (J. Hansen and Kuchling, 2017). Blowfish implementations employ 16 rounds of encryption and are not vulnerable to known-plaintext attacks on weak keys, but a reduced-round variant of Blowfish is known to be vulnerable. Therefore, experts recommend to use AES (Barker, 2020), a symmetric encryption algorithm with a larger block size and stronger security features.

R-02. AES is widely considered as a secure symmetric encryption algorithm, but the choice of encryption mode is crucial for achieving maximum security. ECB (Electronic Codebook) mode is not secure when there is more than one block to encrypt,[3] and CBC (Cipher Block Chaining) mode is secure for non client–server scenarios (Vaudenay, 2002). Indeed, CBC mode is not recommended for client–server scenarios as it requires careful IV management, proper padding, secure key management, and separate mechanisms for data integrity and authentication. Due to our limited knowledge of the specific context in which code snippets are utilized, we consider any encryption performed with ECB or CBC modes as a security risk.[4]

R-03. To ensure a robust initialization of the cipher-key, it is crucial to adhere to the guidelines outlined in R-03-a to R-03-g. The use of static or constant keys and salts for key derivation is not permitted at all (R-03-a and R-03-b). Instead, it is necessary to employ at least 64 bits of salt (R-03-c) and a minimum of 1000 iterations for key derivation (R-03-d). Hard-coded passwords (R-03-e) as well as weak random functions (R-03-f) or weak algorithms (R-03-g) for generating secret keys must be avoided,

R-04. Proper initialization of an Initialization Vector (IV) is crucial for ensuring the security of AES encryption. To this end, R-04-a advises against using a static IV that may be predictable and void its purpose. There has to be a dynamic seed and proper randomness for IV generation (R-04-b) to ensure that each IV is unique and unpredictable for every encryption operation.

R-05. When loading a keystore, it is important to ensure that the password is retrieved from a secure external source, such as a database or a file. Using a constant and non-null password value when loading a keystore via an input stream can introduce security risks.

3.2.2. Manual investigation

Our manual analysis of the posts in the sample dataset indicated that 82%, *i.e.,* 327 posts, are afflicted with at least one security risk, and only 17 posts had an accepted answer with a secure solution.[5] In particular, we found a total of 559 security violations out of which 368 violations were present in 209 posts with an accepted answer (*i.e.,* solved posts). Table 5 presents an overview of these risks, which we discuss below.

R-01. There were a total of 34 posts that adopted a weak algorithm. Out of these posts, 14 of them were solved posts, but none of them offered a secure solution.

R-02. We identified 265 posts that used an insecure mode for encryption, 192 of them were solved posts. There were 127 posts that employed the ECB encryption mode. Regrettably, among the solved posts, only 7 provided secure solutions. We also found 164 posts utilizing CBC encryption mode (R-02-b), which is considered a bad practice for client–server scenarios due to its lack of integrity protection. Unfortunately, only 8 of the solved posts offered secure solutions, considering integrity.

R-03. We observed static keys in 100 posts. Merely 6 solved posts presented secure solutions. There were 20 posts that used static salts for key generation from passwords. We found 4 posts with fewer iterations than 1000, and 25 posts that utilized hard-coded passwords. Only 4 solved posts provided secure solutions. Finally, in 33 posts, developers employed weak algorithms for key generation, and among solved posts, only 8 accepted answers offered a secure solution.

R-04. We found static initialization vectors (IV) in 48 posts. Among the solved posts, only 5 of them suggested a secure solution. We also found 1 post with badly derived IV.

R-05. We found only one violation instance of this rule.

[3] Using AES without specifying an encryption mode is functionally equivalent to utilizing AES in ECB mode.

[4] CBC is secure insofar as it is designed to resist chosen-plaintext attacks, which it effectively does. However, CBC does not defend against chosen-ciphertext attacks, making it a bad practice for client/server scenarios.

[5] The term "secure solution" refers to either a warning that highlights the risk or a code snippet that patches the issue in an accepted answer.

Table 5
Security violations in sample posts.

Rule-ID	#Solved Posts	#Pending Posts	#Total
R-01	14	20	34
R-02-a	86	41	127
R-02-b	106	58	164
R-03-a	73	27	100
R-03-b	12	8	20
R-03-c	0	0	0
R-03-d	4	0	4
R-03-e	17	8	25
R-03-f	1	1	2
R-03-g	18	15	33
R-04-a	36	12	48
R-04-b	1	0	1
R-05	0	1	1

Table 6
Insecure patterns and bad practices (bad practices are shown in orange color, and manual steps are documented in blue color preceded by an → symbol.)

RuleID	RegEx (PCRE2)
R-01	Cipher\.getInstance\((“|&quot)(DES|DESede|RC2|RC4|RC5|Blowfish|chacha20)
R-02-a	Cipher\.getInstance\“(?:AES|DES|DESede|RC2|RC4|RC5|Blowfish|chacha20)(?:\/ECB)?”|Cipher\.getInstance\ (&quot;(?:AES|DES|DESede|RC2|RC4|RC5|Blowfish|chacha20)(?:\/ECB)?&quot;
R-02-b	Cipher\.getInstance\(“(?:AES|DES|DESede|RC2|RC4|RC5|Blowfish|chacha20)\/CBC|Cipher\.getInstance\(&quot; (?:AES|DES|DESede|RC2|RC4|RC5|Blowfish|chacha20)\/CBC
R-03-a	(?:key\s* = |secret\s*)(?:"|&quot;|{|new byte\[\]\{)|SecretKeySpec\(“
R-03-b	PBEKeySpec\(\s*\S+\s*,\s*"\S+"\s*,\s*\S+\s*|PBEParameterSpec\(“|salt\s*(?:“|&quot;|{|new byte\[\]\{)
R-03-c	(salt\s*=|PBEKeySpec\(|PBEParameterSpec\() → check the size of salt.
R-03-d	PBEKeySpec\(\s*\S+\s*,\s*\S+\s*,[1-9]\d{0,2}|PBEParameterSpec\(\s*\S+\s*,[1-9]\d{0,2}|salt\s*= → examine if the Iteration Count value is less than 1000.
R-03-e	(?:password\s*|pass\s*)(?:“|&quot;)|PBEKeySpec\("
R-03-f	kgenerator\.init\(\d+,\s*\w+\) → assess how the random value, *i.e.,* the second parameter, is generated.
R-03-g	SecretKeyFactory\.getInstance\((?:“PBEWithMD5AndDES”|“PBKDF2WithHmacSHA1”)|MessageDigest\ .getInstance\(“SHA-1”\N*\n*\N*\n*\N*.digest\(Key
R-04	(IV\s*|InitVector\s* =|InitializationVector\s* =|IvParameterSpec\()\s*(?:“|&quot;|{|new byte\[\]\s*{)
R-05	KeyStore\.load\(→ examine whether the password, *i.e.,* the second parameter, is hard-coded.

3.2.3. Large-scale analysis

We collected code snippets associated with each security violation during our manual investigation. These snippets helped us to develop a set of regular expressions, listed in Table 6, to locate security violations in the posts. We conducted a large-scale investigation of security issues by applying these expressions to the posts in our full dataset. These expressions could locate violations for rules R-01, R-02-a, R-02-b, and R-03-g, but manual inspection was necessary for R-03-c, R-03-d, R-03-f, and R-05. For instance, R-03-d RegEx only filtered posts that might contain information about the iteration size for key generation, and we performed a subsequent manual check to identify the actual violations. We also inspected the results for R-03-a, R-03-b, R-03-e, and R-04-a that locate constant values. This was to ensure that the identified constants were intended for use within the cipher object. For instance, when encountering a hardcoded password, it is crucial to confirm that it is intended for use as a password-based key generation rather than being applicable to other scenarios. Consider post ID 28519570 as an example, where the specified password has nothing to do with key generation.

We found a total of 5305 issues in 3174 posts, showing that *i.e.,* 92% of posts suffer from on average 1.7 issues. Table 7 lists the distribution of violations in the full dataset. These issues were mostly present in questions, but in 234 posts, 266 issues were introduced in the accepted answers.

We found 629 posts (300 solved) that used weak encryption algorithms. Nevertheless, of 2797 posts (1406 solved) that utilized the recommended AES algorithm for symmetric encryption, the encryption modes were not secure in 2425 posts (1232 solved), weakening the advantage of AES. In particular, 39% were ECB and 48% were CBC modes. Fig. 4 illustrates the share of encryption modes in each year. It seems that the adoption of ECB is decreased in recent years, whereas GCM is increased. Nevertheless, CBC has remained almost constant.

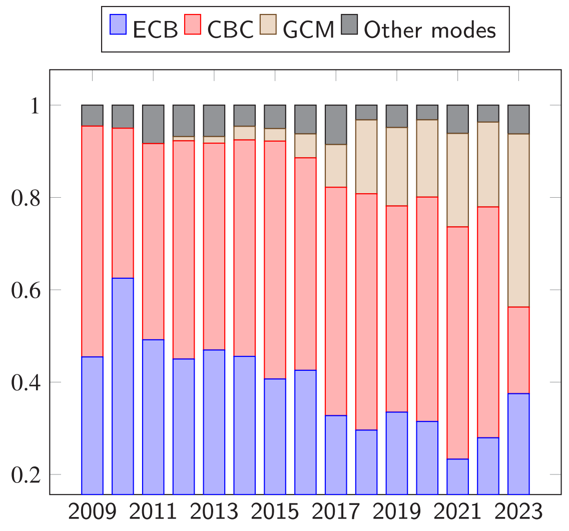


Fig. 4. Share of encryption modes for AES.

Table 7
Security violations in the full dataset.

Rule-ID	#Solved Posts	#Pending Posts	#Total
R-01	300	329	629
R-02-a	717	760	1,477
R-02-b	804	747	1,551
R-03-a	382	356	738
R-03-b	60	51	111
R-03-c	0	4	4
R-03-d	6	15	21
R-03-e	90	92	182
R-03-f	1	3	4
R-03-g	161	109	270
R-04-a	174	140	314
R-05	2	2	4

We found issues during the initialization of cipher keys in 998 posts (494 solved). In particular, using static or constant keys was a common bad practice among programmers, with 738 posts containing this violation (382 solved). There were also 270 posts (161 solved) in which a secret key was generated from a password using insufficiently secure algorithms. The "PBKDF2WithHmacSHA1" algorithm, which is a bad practice, was used in 197 posts (122 solved). "PBEWithMD5AndDES" which is insecure, was used in 11 posts (7 solved). Additionally, the hash function with "SHA1" which is insecure and the worst choice, was used in 60 posts (31 solved).

In 182 posts (90 solved) developers used hard-coded passwords. We found 314 posts (174 solved) that violated the rule for the initialization of cipher initialization vectors (IVs).

The number of security violations in the symmetric encryption posts on the StackOverflow website is significant, making it a misleading and dangerous information source, especially for novices.

3.3. ChatGPT analysis

With the advancement in generative AIs, developers are increasingly consulting them as a development assistant. We examined whether ChatGPT can be relied upon from a security perspective to answer developer questions in 100 StackOverflow posts.

We considered a ChatGPT response as good as an accepted answer if it allowed us to easily reach the solution, similarly to the accepted answers on StackOverflow. When we provided ChatGPT with the exact questions from StackOverflow posts. We observed that 67 responses were relatively as good as accepted answers, but 33 responses were general and not to the point. In terms of security, only three responses were free from violations. In particular, ChatGPT corrected 5 violations in 3 posts but introduced 4 new violations as well.

We observed a notable improvement when for each question we explicitly asked ChatGPT to provide a "secure" solution. Particularly, ChatGPT provided secure solutions for 42 posts (58 remained problematic).

For each security violation that was still present in the ChatGPT answer, we explicitly asked ChatGPT to evaluate the security of the line associated with that violation. This resulted in the resolution of violations in 26 more posts. Nevertheless, 32 posts were still problematic mostly because ChatGPT considered CBC as a secure encryption mode and even in an insecure scenario did not provide any warning.

Security violations may exist in the code examples and hints provided by ChatGPT. Users should exercise caution and follow a line-by-line security check to clear any doubts.

We investigated whether other prompts can help to identify violations without pinpointing the exact problematic line. Even the prompts such as "provide a code example and make sure that every line is secure" or "examine your code example line by line and make sure that each line is secure" or "examine your code example and ensure that each line is secure for production" did not help to address security issues and a manual line-by-line analysis is still superior.

We noted that when ChatGPT was prompted to provide a "secure example for **production**" the outcome did not change except providing warnings against hard-coded and constant values.

Even providing security guidelines in the prompt to ChatGPT did not result in a completely secure solution. For example, consider the following prompt for post ID 3451670:

"[StackOverflow Question 3451670] -> Please answer this question and provide a secure Java code example. I expect a secure encryption algorithm, a secure encryption mode, and a secure Initialization Vector (IV). Do not use hard-coded secrets, suggest best practices, and make sure that your example provides confidentiality and integrity. Minimize the need for additional questions and ensure that every line in your code example follows this instruction".

```java
public class SecureEncryptionExample {
  public static void main(String[] args) throws
    Exception {
    String username = "bob@google.org";
    String password = "Password1";
    String secretID = "BlahBlahBlah";
    String SALT2 = "deliciously salty";
    // Generate a secure random IV (Initialization
    Vector)
    SecureRandom secureRandom = new SecureRandom();
    byte[] iv = new byte[12];
    secureRandom.nextBytes(iv);
    // Derive a key from the password using a secure
    method
    byte[] key = deriveKey(SALT2, username, password);
    // Create a SecretKey object using the derived key
    SecretKey secretKey = new SecretKeySpec(key, "AES")
    ;
    // Instantiate the cipher with AES in GCM mode
    Cipher cipher = Cipher.getInstance("AES/GCM/
    NoPadding");
    ...
  }
  private static byte[] deriveKey(String salt, String
    username, String password) throws Exception {
    // Concatenate salt, username, and password
    String combined = salt + username + password;
    // Use a secure hash function (SHA-256) to derive a
    key
    MessageDigest md = MessageDigest.getInstance("SHA
    -256");
    byte[] hash = md.digest(combined.getBytes());
    return Arrays.copyOf(hash, 16);
  }
}
```

Listing 2: Part of ChatGPT's answer to post ID 3451670

Listing 2 presents the response.[6] ChatGPT identified the constant key, and it correctly suggested the secure "GCM" encryption mode as a best practice. However, violations such as "static salt for key generation" and "hard-coded password" were in the response without any proper warning.

The suggested code, at line 23, included "SHA-256" hash function for key generation, which is not a best practice. Even worse, ChatGPT fails to address a critical issue with this construction in line 21. There is no domain separation between the three input fields. This means that both ("user1", "password") and ("user", "1password") result in the same derived key, potentially leading to catastrophic consequences. Anyway, when we asked about the security risk at line 6 (*i.e.,* static salt), ChatGPT responded: *"Using a hard-coded string as a salt is generally not considered a good practice for security. Ideally, a salt should be unique for each user and should be randomly generated for each password. Using a constant salt for all users could potentially make the system more vulnerable to certain types of attacks, such as rainbow table attacks"*. It then provided the code example in Listing 3 as a replacement.

```java
// Generate a secure random salt
SecureRandom secureRandom = new SecureRandom();
byte[] salt = new byte[16];
secureRandom.nextBytes(salt);
// Convert the salt to a Base64-encoded string for
    storage
String SALT2 = Base64.getEncoder().encodeToString(salt
    );
```

Listing 3: Part of ChatGPT's answer to post ID 3451670

Subsequently, we asked, *"Are there any other security risks in your provided code?"*, but it failed to detect hard-coded password. But in this step, it correctly suggested a key derivation function (KDF) instead of a hash function for generating the password-base key. Upon a direct inquiry about the line where a hard-coded password existed, it offered a secure suggestion.

4. Discussion

We reflect on our findings about developers' challenges in symmetric encryption, security violations, and ChatGPT's support.

4.1. Developer challenges

The most common root causes for developer issues were in key initialization (121), iv initialization (106), padding (89), encoding (54), and encryption mode (49). Research suggests that the most significant reasons for these challenges among developers could stem from a lack of background knowledge (Nadi et al., 2016; Patnaik et al., 2019), inadequate documentation (Nadi et al., 2016; Hazhirpasand et al., 2020; Patnaik et al., 2019; Acar et al., 2017), usability issues (Nadi et al., 2016; Green and Smith, 2016), low adoption of static analysis tools among developers (Hazhirpasand and Ghafari, 2021b), and finally limitations in static analysis tools for detecting misuses (Zhang et al., 2023; Afrose et al., 2023; Ami et al., 2022).

Security concerns were naturally predominant in the posts, and starting from 2020, we observed a consistent increase in concerns related to interoperability. We were unable to locate any posts discussing compatibility concerns in the past two years (*i.e.,* 2022 and 2023), showing an improvement in this domain.

We came across 128 exceptions during the inspection of sample posts. These exceptions, listed in Table 8, initially appeared to be associated with padding and key-related issues, but further investigations revealed that the root causes may be different. We found 61 cases where a `BadPaddingException` occurred. However, only 19 cases were directly related to padding problems. Most notably, 17 exceptions occurred during the encryption and decryption stages, primarily due to incorrect encoding. We identified 14 cases with key-related issues, nine cases involving initialization vector (IV) problems, and two other problems. In 45 instances where an `InvalidKeyException` occurred, only 26 exceptions were directly linked to key initialization problems. Eight exceptions were thrown in the context of initialization vector (IV) issues, while the remaining 11 exceptions were associated with other problems.

General exception messages are confusing as observed in 25% of sample posts. Precise and informative exceptions can help developers to identify root causes.

4.2. Security investigation

Through our manual inspection of 400 posts, we found a total of 559 security violations in 327 posts. From 251 posts with an accepted answer, only 32 posts were free from such violations. In other words, from every 8 posts, only one is reliable. Nevertheless, posts might contain short or incomplete code snippets requiring users to consult other posts which might introduce violations. Even in 17 posts that included a direct question about security, only two received an accepted answer without any violation, and in two posts, new violations (R-03-e, R-04-a) were introduced in the accepted answer.

In our large-scale investigation of 3426 posts, we found 5305 issues in 3174 posts. We compared the distribution of security violations in the sample dataset versus the full dataset. We observed an increase, approximately two-fold, in the prevalence of weak algorithms in the full dataset. This could be attributed to a decline in the adoption of weak algorithms in recent years especially that our sample data was inclined towards recent and popular posts. For the remaining rules, the distribution in the sample data closely mirrors that of the full dataset.

Several violations such as rules related to the presence of constant values may exist in the posts for demonstration purposes and providing working code examples (such as `salt=''1234''` or `key=''passkey''`). However, without a proper warning (which is almost absent in the posts), it is likely that novice programmers who resort to a copy-and-pasting approach, reuse these code examples, and at best, only substitute the constant values with new ones. Even if we exclude so called demonstration mistakes, for instance from 400 sample posts, 312 posts remain with 361 issues. By eliminating R-02-b (*i.e.,* the presence of CBC mode) these numbers were reduced to 180 posts and 197 issues, which are still problematic.

The minimum key length for AES should be 128 bits, but for applications demanding more security, NIST advises a 256-bit key (Barker, 2020). Nevertheless, keys with a size of 128 were dominated for a long time. From 2013 onward, we observed a declining trend in 128-bit keys and an upward trajectory in 256-bit keys. In 2018, the distributions became equal. This might be attributed to the release of JDK 9 that enabled developers to adopt stronger cryptographic algorithms by default, whereas in earlier versions they had to install "JCE Unlimited Strength Jurisdiction Policy Files" manually. It may also be driven by concerns over the quantum computing threat, which theoretically diminishes key sizes by half. From 2019, a sharp decline in the use of 128-bit keys and a surge in 256-bit keys were evident. In addition, we observed an increase in the adoption of the more secure GCM mode in recent years, notably from 2018. There were a total of 198 posts that employed GCM, and interestingly, only 30 posts (*i.e.,* 15%) included violations.

[6] https://chat.openai.com/share/8ce9eb6e-88b7-44dc-807a-51d04e7d446c

Table 8
Top three recurrent exceptions.

Exception Type	#	Exception Message (#)
BadPaddingException	61	Given final block not properly padded (36)
		pad block corrupted (13)
InvalidKeyException	45	Illegal key size (14)
		Invalid AES key length (9)
		Key length not 128/192/256 bits (6)
		Parameters missing (6)
		No installed provider supports this key (3)
		Wrong Algorithm (3)
IllegalBlockSizeException	19	Input length must be multiple of 16 (10)
		Last block incomplete in decryption (3)

Furthermore, in 50 posts, we observed best practices such as "PBKDF2WithHmacSHA256" or "PBKDF2WithH-macSHA512" for generating a secret key from a password. However, the bad practice "PBKDF2WithHmacSHA1" was a more common practice, occurring in 197 posts. We also observed the adoption of insecure algorithms such as "PBEWithMD5AndDES" in 11 posts and "SHA-1" hash function for key generation in 60 posts. There were 28 posts that employed "SHA256" and "SHA512" hash functions for key generation which is secure but not the best practice. We did not observe any additional noteworthy security best practices.

Finally, we looked at the profiles of users who contributed to accepted answers. We did not observe a better performance of users with a crypto badge than those with a security badge. Likewise, reputation was not a good indicator of security. In particular, 15 users who had a security-related badge (passwords, security, spring-security) contributed to 90 issues in 84 posts, 27 users with a cryptography-related badge (aes, aes-gcm, cryptography, jce, encryption) made 199 issues in 158 posts, and 192 users with a reputation of at least 2000 made 410 issues in 336 posts. This observation is inline with what previous work has shown about the success of developers in cryptography (Hazhirpasand et al., 2019).

There are tons of StackOverflow posts that have security implications but are outdated. Generative AI assistants such as ChatGPT can be effective in flagging such content and improving developer awareness.

4.3. ChatGPT support

ChatGPT responses were correct, but in 33 cases tended to be overly general and missed the mark in addressing the question directly. Such responses may not be particularly helpful for beginners. A notable example highlighting this issue was when a developer posted a question (post ID 75266913) seeking help with an error in code generated by ChatGPT. In the following, we discuss ChatGPT's performance through the lens of security.

ChatGPT correctly detected "DES" as an insecure encryption algorithm, However, it suggested "3DES" as a good choice although it is deprecated. For instance, as we observed in response to post ID 22951606,[7] it suggested to use "3DES with a 192-bit key", and included a note that *"If you have the option, consider using AES with a 256-bit key for even stronger security"*.

ChatGPT correctly flagged ECB as an insecure encryption mode, but in some cases, *e.g.,* post ID 10759392,[8] it suggested to replace it with CBC, which could be problematic. Indeed, While CBC itself is a secure encryption mode, it lacks integrity protection, making it insecure for client/server scenarios. However, ChatGPT may still recommend CBC as a secure choice, independent of its context. For instance, in response to post ID 18291987 concerning a client/server scenario, ChatGPT suggested CBC as a secure choice, despite its unsuitability in that context.[9] Nevertheless, in several cases, ChatGPT suggested using GCM instead of CBC for additional security.

ChatGPT did not always detect the use of constant values, even though we asked for a secure code example. It did catch this violation when we explicitly asked about the security of the exact line that included hard-coded secrets. Nevertheless, ChatGPT mostly suggests to replace the constant values, *e.g.,* "with your own secrets", which may still yield hard-coded secrets in the code by novices. For example, in answer to post ID 992019, ChatGPT suggested a hard-coded password but also included this warning: *"You should use a strong passphrase, as a weak passphrase can be easily broken by an attacker. You should also make sure to use a secure method of storing the passphrase, as it is the key to your encrypted data"*..

ChatGPT flagged "SHA-1" hash function for key derivation as insecure. However, we observed instances where it suggested using the "SHA-256" hash function for key generation (e.g., post ID 992019).[10] Although "SHA-256" is secure, best practices recommend to adopt a key derivation function, providing additional security measures like salting and iteration count to slow down the key derivation process, making it more resistant to brute-force attacks. ChatGPT also correctly flagged "PBEWithMD5AndDES" as a non-secure password-based key generation technique, but it recommended "PBKDF2WithHmacSHA1" as a secure option, whereas it is not a good practice. In some cases, ChatGPT suggested "PBKDF2WithHmacSHA256" for higher security.

ChatGPT was not consistent in detecting violations of the number of iterations. For example, for post ID 4202499,[11] it suggested to increase the iterations to at least 1000, whereas it missed the same violation in post ID 20888851.[12] Indeed, We realized that ChatGPT's response, even to the same question, might change over time. For example, when we queried about post ID 18291987,[13] it correctly suggested to use GCM mode for the chat application. However, when we posed the same question at a later time,[14] it suggested to use CBC, which is certainly not a proper choice in a client–server scenario.

[7] https://chat.openai.com/share/2d03613d-073e-4aaa-a644-9e2527c4dd75
[8] https://chat.openai.com/share/c1150457-1c39-4019-94a1-152b93e53fad
[9] https://chat.openai.com/share/8149c107-659c-4995-a1f7-c9fe7d487a3c
[10] https://chat.openai.com/share/9e4e0583-f5e4-4cce-84e7-8161da2a70ae
[11] https://chat.openai.com/share/d5998eeb-7da5-416f-8b18-19865f7f5e71
[12] https://chat.openai.com/share/9036dbd0-07dc-474a-8655-4c6e9c91437d
[13] https://chat.openai.com/share/571da441-3029-473d-8c0b-c26dec4a2be3
[14] https://chat.openai.com/share/8149c107-659c-4995-a1f7-c9fe7d487a3c

ChatGPT is helpful but developers should know its limits. It does not know every best practice and does not treat the same security violation consistently. Therefore, ChatGPT cannot replace human expertise, and developers should remain alert throughout a ChatGPT session.

5. Threats to validity

In this section, we provide a summary of potential threats to the validity of this study, along with the strategies implemented to mitigate these threats (Wohlin et al., 2012).

5.1. Threats to internal validity

There are several threats to internal validity that may potentially affect the soundness of our study.

Data Collection: Our data collection approach for symmetric encryption posts might not be exhaustive. We primarily relied on the `Cipher.getInstance` pattern and did not investigate posts that may be relevant but lack the specified code snippet. To expand our coverage, we also searched for posts tagged with "java" and containing the term "symmetric" in the question, resulting in 346 additional posts. However, a manual review revealed that most of these posts were unrelated to JCA. Furthermore, we initially excluded 73 questions with "java" and "symmetric" tags, but further examination suggested that they might contain suitable code snippets for security analysis. Additionally, 228 posts contained the "Cipher.getInstance" pattern in answers instead of questions and lacked accepted answers. Despite these challenges, we believe our data collection is comprehensive. While we may have missed a few posts with pertinent code snippets, our dataset largely aligns with our research objectives.

The inclusion of posts without accepted answer enabled a more comprehensive view of developer challenges including the ones that are still open (pending posts). Nevertheless, security issues in solved posts have a higher impact than issues in pending posts. Therefore, we presented results for pending and solved posts separately.

Categorization Step: Human bias and errors are likely in manual investigations. Nevertheless, we employed a review strategy where each post was checked by at least two people and utilized a majority voting to clear disagreements.

Security Investigation: During the manual investigation, we may have missed some security risks due to human errors. To mitigate this, we implemented a double-checking process and used predefined rules from our base papers. It is possible that other security risks exist that we may have missed, but we strived to gather all relevant rules based on existing research.

During a large-scale study focusing on rules related to constant value violation, there is a possibility that our regular expressions might not have been able to identify constant or hard-coded values that were passed through variables. To address this concern, we implemented a strategy that involved using regular expressions to limit the number of posts we processed automatically, while manually reviewing the remaining posts to minimize the likelihood of missing any security issues.

Additionally, it is important to acknowledge that our collection of regular expressions may not cover all possible cases comprehensively. For instance, there is a chance that we might overlook insecure code snippets related to the constant key, as we primarily considered keywords such as `key` and `secret` appearing at the end of variable names. However, it is worth noting that, during our manual analysis, we observed a high likelihood of users using variable names containing these keywords.

Furthermore, including a large number of keywords in our regular expressions can make the detection process more challenging. Therefore, while our approach has its limitations, we have taken steps to mitigate these issues and enhance the overall accuracy of our security assessment.

ChatGPT Analysis: In our security analysis of ChatGPT responses, we acknowledge that human error and bias can impact the results. To address this concern, we implemented a double-checking system to enhance the accuracy of our findings. Additional reviews were conducted to minimize inaccuracies and improve overall precision.

5.2. Threats to external validity

Threats to external validity stem from considerations of the generalizability of our study's results. It is crucial to acknowledge that our empirical investigation concentrated specifically on Java-related queries on StackOverflow. Hence, the extent to which our findings can be applied to other programming languages or platforms may be constrained. Furthermore, we investigated security risks based on the presence of insecure patterns, which may not uncover all conceivable vulnerabilities.

In evaluating the efficacy of LLM-generated code in addressing cryptographic challenges and ensuring the security of the generated code, we utilized the GPT-3.5 model as a representative of widely used models (chosen for its availability and popularity). It is important to recognize that this choice may limit the generalizability of our results to other models. To address this limitation, we plan to explore other models in the future, facilitating a comparative analysis that can yield more comprehensive insights into the security, correctness, and completeness of AI-generated code

6. Related work

We discuss related work in three key areas: code snippets security, cryptography challenges and misuses, and finally, evaluation of AI-generated code.

6.1. StackOverflow code snippets security

Several studies have examined the security of code snippets from StackOverflow.

Verdi et al. (2022) revealed a concerning trend where a substantial number of insecure C++ code snippets had been transferred into GitHub projects. Firouzi et al. (2020) studied C# code snippets that include unmanaged code and found 67 code snippets with dangerous functions that can introduce vulnerability (e.g., buffer overflow) if not used with caution.

André et al. (2022) investigated the security issues that developers encounter in WebAssembly, and they found that the topmost issues attributed to authentication in Blazor WebAssembly.

Meng et al. (2018) examined code snippets for Java security. They observed an increasing trend in the adoption of third-party security frameworks like Spring Security for authentication and authorization. Also, they discovered that many accepted answers contained security flaws such as weak hash functions (MD5), SSL/TLS compromises, and disabled CSRF protection.

Rahman et al. (2019) analyzed 529,054 Python code blocks from Stack Overflow posts. They identified 3685 code blocks that exhibited at least one of the six critical insecure coding practices, namely code injection, cross-site scripting, use of insecure ciphers, insecure communications, race conditions, and insecure data serialization.

Fischer et al. (2017) employed machine learning techniques, training a model with 1360 security-related code snippets sourced from StackOverflow. This model was subsequently applied to 3834 code snippets, revealing that 30.28% of them were identified as insecure.

6.2. Cryptography challenges and misuses

Several studies have explored the difficulties faced by developers in dealing with cryptography (Nguyen et al., 2017; Gutmann, 2002; Shuai et al., 2014; Rahaman et al., 2019).

Hazhirpasand et al. (2021b) clustered cryptography questions on StackOverflow, inspected a subset of them, and discussed why developers struggle in this domain. They also analyzed JCA misuses in 489 open-source projects on GitHub and found that only 15% of repositories were free from misuse, leaving 85% susceptible to security issues (Hazhirpasand et al., 2020). The investigation of GitHub projects also revealed that approximately 64% of cryptographic solutions in each project were not secure (Hazhirpasand et al., 2019).

Gajrani et al. (2017) revealed a troubling statistic indicating that a significant 90% of applications available across diverse app stores are susceptible to exploitation due to cryptographic vulnerabilities. Lazar et al. (2014) conducted a systematic study of cryptographic vulnerabilities in practice. Hazhirpasand and Ghafari (2021a) investigated cryptography vulnerability reports on the HackerOne bug bounty platform.

Several studies (Piccolboni et al., 2021; Krüger et al., 2017; Zhang et al., 2022) introduced tools for misuse detection. However, the adoption of static analysis tools is low among developers (Hazhirpasand and Ghafari, 2021b). To ease the adoption of static analysis tools for mainstream developers, Mehdi Pourhashem Kallehbasti and Ghafari (2023) developed NASRA (NAturalistic Static pRogram Analysis), a framework that enables developers to define static program analyses in natural language, and they showcased how NASRA can be applied to uncover cryptography misuses in Java programs. Nonetheless, existing tools are not able to catch every mistake. Braga et al. (2017) found that coverage of static tools for finding cryptography missuses is far from good. Zhang et al. (2023) conducted a comprehensive review of six crypto misuse detection tools by applying them to 200 Apache projects. They found that there is no single tool being universally superior and that improvement in inter-procedural analysis and context awareness are necessary to effectively find misuses. Ami et al. (2022) introduced a mutation testing framework to systematically assess the effectiveness of crypto-API misuse detectors.

Afrose et al. (2023) developed detailed benchmarks and executed several vulnerability detection tools for the comparison of their effectiveness and reported their findings. Overlooked issues include difficulties in resolving parameter values, insecure initialization vectors, insecure random number generation, and insufficient key lengths in cryptographic key generation. Finally, Kafader and Ghafari (2021) developed FluentCrypto, an API designed to abstract away the low-level complexities inherent in utilizing the Node.js native cryptography API.

6.3. Evaluation of AI-generated code

The recent advancements in AI assistant tools have motivated researchers to examine AI-generated code. Fu et al. (2023) analyzed 435 code snippets generated by Copilot in public GitHub projects and found that 35.8% had Common Weakness Enumeration (CWE) issues across various programming languages. Pearce et al. (2022) studied Copilot's performance in suggesting code related to 89 scenarios that were subject to MITRE's "Top 25" CWEs. They discovered that around 40% of the generated programs include vulnerabilities. Asare et al. (2023) conducted a study on GitHub's Copilot, examining its performance in various C++ and C scenarios. The findings revealed that while Copilot generated insecure code, it was not as bad as human developers in producing insecure code. Sandoval et al. (2023) found that AI assistance in low-level C programming minimally affected security, introducing critical bugs only slightly more often (up to 10%) than in cases without AI, indicating that the use of LLMs does not introduce new security risks.

Perry et al. (2023) studied how individuals utilize an AI code assistant, constructed using OpenAI's Codex, to address security-related tasks across various programming languages. They found that participants who used the AI were more likely to introduce security vulnerabilities, yet they often perceived their insecure solutions as secure. Interestingly, those who invested more effort in crafting their queries to the AI tended to generate more secure solutions.

Kavian et al. (2024) developed LLMSecGuard, an open-source framework designed to enhance code security through the integration of LLMs and static security code analyzers. It uses hints from static security analysis tools to guide LLMs in writing secure code.

We presented the first in-depth investigation of symmetric encryption challenges and security risks. In contrast to previous studies, we also investigated the support of generative AI (i.e., ChatGPT) to fix these issues.

7. Conclusion

We conducted a thorough examination of StackOverflow posts related to symmetric encryption in Java. We delved into developers' challenges and concerns, discovering that they struggle significantly with key and IV generations, as well as padding. We observed that security is a top concern among developers. Nonetheless, we examined each post from a security perspective and discovered that security violations related to symmetric encryption abound, making StackOverflow a risky information source, especially for novices. We evaluated ChatGPT's support in resolving these issues and found that it can be very helpful only if proper interaction is in place. Nevertheless, ChatGPT does not substitute human expertise, and developers should remain alert.

CRediT authorship contribution statement

Ehsan Firouzi: Writing – original draft, Methodology, Investigation. **Mohammad Ghafari:** Writing – review & editing, Supervision, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the link.

References

Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M.L., Stransky, C., 2017. Comparing the usability of cryptographic APIs. In: 2017 IEEE Symposium on Security and Privacy. IEEE, pp. 154–171.

Afrose, S., Xiao, Y., Rahaman, S., Miller, B.P., Yao, D., 2023. Evaluation of static vulnerability detection tools with Java cryptographic API benchmarks. IEEE Trans. Softw. Eng. 49 (2).

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

André, P.M., Stiévenart, Q., Ghafari, M., 2022. Developers struggle with authentication in blazor WebAssembly. In: 2022 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 389–393. http://dx.doi.org/10.1109/ICSME55016.2022.00045.

Asare, O., Nagappan, M., Asokan, N., 2023. Is GitHub's copilot as bad as humans at introducing vulnerabilities in code? Empir. Softw. Eng. 28, 129. http://dx.doi.org/10.1007/s10664-023-10380-1.

Barker, E., 2020. Recommendation for Key Management: Part 1—General (NIST Special Publication 800-57 Part 1 Revision 5). National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA.

Barker, E., Mouha, N., 2017. Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA, URL http://dx.doi.org/10.6028/NIST.SP.800-67r2. Withdrawn on January 01, 2024.

Bhargavan, K., Leurent, G., 2016. Sweet32: Birthday attacks on 64-bit block ciphers in TLS and OpenVpn. URL https://sweet32.info/.

Braga, A., Dahab, R., Antunes, N., Laranjeiro, N., Vieira, M., 2017. Practical evaluation of static analysis tools for cryptography: Benchmarking method and case study. In: 2017 IEEE 28th International Symposium on Software Reliability Engineering. ISSRE, pp. 170–181. http://dx.doi.org/10.1109/ISSRE.2017.27.

Bühlmann, N., Ghafari, M., 2022. How do developers deal with security issue reports on GitHub? In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. SAC '22, pp. 1580–1589. http://dx.doi.org/10.1145/3477314.3507123.

Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J., 2012. Non-Functional Requirements in Software Engineering, vol. 5, Springer Science & Business Media.

Ferguson, N., Schneier, B., Kohno, T., 2011. Cryptography Engineering: Design Principles and Practical Applications. John Wiley & Sons.

Firouzi, E., Sami, A., Khomh, F., Uddin, G., 2020. On the use of C# unsafe code context: An empirical study of stack overflow. In: Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, ESEM '20, http://dx.doi.org/10.1145/3382494.3422165.

Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., Fahl, S., 2017. Stack overflow considered harmful? The impact of copy&paste on android application security. In: 2017 IEEE Symposium on Security and Privacy. SP, pp. 121–136. http://dx.doi.org/10.1109/SP.2017.31.

Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., 2023. Security weaknesses of copilot generated code in GitHub.

Gajrani, J., Tripathi, M., Laxmi, V., Gaur, M.S., Conti, M., Rajarajan, M., 2017. sPECTRA: A precise framework for analyzing CrypTographic vulnerabilities in android apps. In: 2017 14th IEEE Annual Consumer Communications & Networking Conference. CCNC, pp. 854–860. http://dx.doi.org/10.1109/CCNC.2017.7983245.

Green, M., Smith, M., 2016. Developers are not the enemy! the need for usable security APIs. IEEE Secur. Priv. 14 (5), 40–46.

Gutmann, P., 2002. Lessons learned in implementing and deploying crypto software. In: 11th USENIX Security Symposium. USENIX Security 02, USENIX Association, San Francisco, CA, URL https://www.usenix.org/conference/11th-usenix-security-symposium/lessons-learned-implementing-and-deploying-crypto-software.

Hazhirpasand, M., Ghafari, M., 2021a. Cryptography vulnerabilities on HackerOne. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security. QRS, pp. 18–27. http://dx.doi.org/10.1109/QRS54544.2021.00013.

Hazhirpasand, M., Ghafari, M., 2021b. Worrisome patterns in developers: A survey in cryptography. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops. ASEW.

Hazhirpasand, M., Ghafari, M., Krüger, S., Bodden, E., Nierstrasz, O., 2019. The impact of developer experience in using Java cryptography. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, pp. 1–6. http://dx.doi.org/10.1109/ESEM.2019.8870184.

Hazhirpasand, M., Ghafari, M., 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, ACM/IEEE, pp. 1–6.

Hazhirpasand, M., Nierstrasz, O., Ghafari, M., 2021a. Dazed and confused: What's wrong with crypto libraries?. In: 18th International Conference on Privacy, Security and Trust. PST, pp. 1–6.

Hazhirpasand, M., Nierstrasz, O., Shabani, M., Ghafari, M., 2021b. Hurdles for developers in cryptography. In: 2021 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 659–663.

J. Hansen, R., Kuchling, A., 2017. Gnupg frequently asked questions. URL https://www.gnupg.org/faq/gnupg-faq.html#define_fish.

Jalote, P., Murphy, B., Garzia, M., Errez, B., Way, O.R., 2004. Measuring reliability of software products. In: 15th International Symposium on Software Reliability Engineering.

Kafader, S., Ghafari, M., 2021. FluentCrypto: Cryptography in easy mode. In: 2021 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 402–412. http://dx.doi.org/10.1109/ICSME52107.2021.00042.

Kavian, A., Pourhashem Kallehbasti, M.M., Kazemi, S., Firouzi, E., Ghafari, M., 2024. LLM security guard for code. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering. EASE '24.

Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M., 2017. CrySL: Validating correct usage of cryptographic APIs.

Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M., 2019. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. IEEE Trans. Softw. Eng..

Lazar, D., Chen, H., Wang, X., Zeldovich, N., 2014. Why does cryptographic software fail? a case study and open problems. In: Proceedings of 5th Asia-Pacific Workshop on Systems. APSys '14.

Mehdi Pourhashem Kallehbasti, M., Ghafari, M., 2023. Naturalistic static program analysis. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 743–747. http://dx.doi.org/10.1109/SANER56733.2023.00083.

Meng, N., Nagy, S., Yao, D., Zhuang, W., Arango-Argoty, G., 2018. Secure coding practices in Java: Challenges and vulnerabilities. In: 2018 IEEE/ACM 40th International Conference on Software Engineering. ICSE, pp. 372–383. http://dx.doi.org/10.1145/3180155.3180201.

Mindermann, K., Keck, P., Wagner, S., 2018. How usable are rust cryptography apis? In: 2018 IEEE International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 143–154.

Nadi, S., Krüger, S., Mezini, M., Bodden, E., 2016. Jumping through hoops: Why do Java developers struggle with cryptography apis? In: Proceedings of the 38th International Conference on Software Engineering. pp. 935–946.

Nguyen, D.C., Wermke, D., Acar, Y., Backes, M., Weir, C., Fahl, S., 2017. A stitch in time: Supporting android developers in WritingSecure code. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. CCS '17, pp. 1065–1077.

Oracle, 2021. Java cryptography architecture (JCA) reference guide. URL https://docs.oracle.com/en/java/javase/17/security/java-cryptography-architecture-jca-reference-guide.html.

Patnaik, N., Hallett, J., Rashid, A., 2019. Usability smells: An analysis of developers' struggle with crypto libraries. In: Fifteenth Symposium on Usable Privacy and Security. usenix, pp. 245–257.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R., 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In: 2022 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 754–768.

Perry, N., Srivastava, M., Kumar, D., Boneh, D., 2023. Do users write more insecure code with AI assistants? In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. CCS '23.

Piccolboni, L., Di Guglielmo, G., Carloni, L.P., Sethumadhavan, S., 2021. Crylogger: Detecting crypto misuses dynamically. In: 2021 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1972–1989.

Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., 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, pp. 2455–2472.

Rahman, A., Farhana, E., Imtiaz, N., 2019. Snakes in paradise? Insecure Python-related coding practices in stack overflow. In: Proceedings of the 16th International Conference on Mining Software Repositories. MSR '19, pp. 200–204.

Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., Dolan-Gavitt, B., 2023. Lost at C: A user study on the security implications of large language model code assistants. In: 32nd USENIX Security Symposium. USENIX Security 23, USENIX Association, Anaheim, CA, pp. 2205–2222, URL https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval.

Shuai, S., Guowei, D., Tao, G., Tianchang, Y., Chenjie, S., 2014. Modelling analysis and auto-detection of cryptographic misuse in android applications. In: 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing. pp. 75–80. http://dx.doi.org/10.1109/DASC.2014.22.

Stackexchange, 2023. Stack exchange directory listing-internet archive. URL https://archive.org/details/stackexchange_20230308.

Vaudenay, S., 2002. Security flaws induced by CBC padding — Applications to SSL, IPSEC, WTLS.... In: Knudsen, L.R. (Ed.), Advances in Cryptology — EUROCRYPT 2002. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 534–545.

Verdi, M., Sami, A., Akhondali, J., Khomh, F., Uddin, G., Motlagh, A.K., 2022. An empirical study of C++ vulnerabilities in crowd-sourced code examples. IEEE Trans. Softw. Eng. 48 (5), 1497–1514.

Wetzels, J., Dos Santos, D., Ghafari, M., 2023. Insecure by design in the backbone of critical infrastructure. In: CPS-IoT Week '23, Association for Computing Machinery, New York, NY, USA, pp. 7–12. http://dx.doi.org/10.1145/3576914.3587485.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer.

Zhang, Y., Kabir, M.M.A., Xiao, Y., Yao, D., Meng, N., 2023. Automatic detection of Java cryptographic API misuses: Are we there yet? IEEE Trans. Softw. Eng. 49 (1), 288–303. http://dx.doi.org/10.1109/TSE.2022.3150302.

Zhang, Y., Xiao, Y., Kabir, M.M.A., Yao, D., Meng, N., 2022. Example-based vulnerability detection and repair in Java code. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. pp. 190–201.

Ehsan Firouzi is a Ph.D. candidate in the Secure Software Engineering (SSE) research group of TU Clausthal, Germany. Ehsan is interested in mining software repositories for security applications. He received his M.Sc. in Information Technology Engineering – Secure Computing from Shiraz University in 2020.

Mohammad Ghafari is a Professor of Software Engineering in TU Clausthal, where he leads the Secure Software Engineering (SSE) group. Mohammad's research focus is on developing tools and techniques that facilitate secure software development. He obtained his Ph.D. in Software Engineering from Politecnico di Milano in 2015.