Beyond Rules: How Large Language Models Are Redefining Cryptographic Misuse Detection

Zohaib Masood and Miguel Vargas Martin
Ontario Tech University, Oshawa, Canada

Keywords: Cryptographic Misuse Detection, Large Language Models, Static Analysis.

Abstract: The use of Large Language Models (LLMs) in software development is rapidly growing, with developers increasingly relying on these models for coding assistance, including security-critical tasks. Our work presents a comprehensive comparison between traditional static analysis tools for cryptographic API misuse detection—CryptoGuard, CogniCrypt, and Snyk Code—and the LLMs—GPT, Llama, Claude, and Gemini. Using benchmark datasets (OWASP, CryptoAPI, and MASC), we evaluate the effectiveness of each tool in identifying cryptographic misuses. Our findings show that GPT 4-o-mini surpasses current state-of-the-art static analysis tools on the CryptoAPI and MASC datasets, though it lags on the OWASP dataset. Additionally, we assess the quality of LLM responses to determine which models provide actionable and accurate advice, giving developers insights into their practical utility for secure coding. This study highlights the comparative strengths and limitations of static analysis versus LLM-driven approaches, offering valuable insights into the evolving role of AI in advancing software security practices.

1 INTRODUCTION

Protecting sensitive data on digital devices from eavesdropping or forgery relies primarily on cryptography. To ensure effectiveness in protection, the cryptographic algorithms employed must be conceptually secure, implemented accurately, and utilized securely within the relevant application. Despite the existence of mature and still secure cryptographic algorithms, numerous studies have pointed out that application developers face challenges in utilizing the Application Programming Interfaces (APIs) of libraries that incorporate these algorithms. As an illustration, Lazar et al. (Lazar et al., 2014) examined 269 vulnerabilities related to cryptography and discovered that merely 17% are associated with flawed algorithm implementations, while the remaining 83% stem from the misuse of cryptographic APIs by application developers. Additional research indicates that around 90% of applications utilizing cryptographic APIs include at least one instance of misuse (Chatzikonstantinou et al., 2016; Egele et al., 2013).

Software designers and developers need to tackle this security flaw by ensuring that their applications encrypt the sensitive data they handle and store. Despite the availability of educational materials (Graff and Wyk, 2003) aimed at raising awareness and providing guidance on secure code development, many developers remain unaware of security considerations (Xie et al., 2011). Training initiatives are not universally received by programmers (Xie et al., 2011), and security is frequently treated as a secondary, desired goal (Whitten, 2004), rather than a mandatory one, depending on the perceived risk and criticality of the developed application. Typically, developers prioritize meeting functionality and time-to-market requirements as their primary goals.

In exploring the factors contributing to this prevalent misuse, researchers previously triangulated findings from four empirical studies, including a survey involving Java developers with prior experience using cryptographic APIs (Nadi et al., 2016). Their findings reveal that a significant majority of participants encountered difficulties in utilizing the respective APIs. Several other tools (Rahaman et al., 2019; Krüger et al., 2017; Zhang et al., 2019; Kafader and Ghafari, 2021) have been developed to detect cryptographic misuses, however, the robustness of these tools have been still in question (Zhang et al., 2023; Ami et al., 2022). In addition to it, novice developers have started adopting AI-assisted tools to code their programming problems. Recent advancements encompass Github's Copilot (GitHub AI Pair Programmer, nd), DeepMind's AlphaCode (Li et al., 2022),

Masood, Z. and Martin, M. V.
Beyond Rules: How Large Language Models Are Redefining Cryptographic Misuse Detection.
DOI: 10.5220/0013524100003979
In *Proceedings of the 22nd International Conference on Security and Cryptography (SECRYPT 2025)*, pages 179-194
ISBN: 978-989-758-760-3; ISSN: 2184-7711

Amazon's Q Developer (Amazon Q Developer, nd), Tabnine (Tabnine, nd), Google's Gemini (Gemini, nd), Meta's Llama (Llama, nd), Anthropic's Claude (Claude, nd), Qwen's QwenLM (QwenLM, nd) and Open AI's ChatGPT (ChatGPT, nd)—nine systems capable of translating a problem description into code. Prior results suggest that Open AI's ChatGPT reliably produces Java programming solutions known for their elevated readability and well-organized structure (Ouh et al., 2023).

Prior research has primarily tested Large Language Models (LLMs) using limited benchmark datasets, offering an initial understanding of their performance. However, a comprehensive study evaluating LLM effectiveness across a broader range of benchmarks is still lacking, and no prior work has closely examined the quality of LLM responses, particularly their accuracy and actionability for practical use. This study addresses these gaps by analyzing LLM performance across diverse datasets and introducing a framework to assess response quality. We assess LLM efficacy in identifying cryptographic misuses and compare it with state-of-the-art (SOTA) static cryptographic analysis tools from literature, such as CogniCrypt (Krüger et al., 2017), CryptoGuard (Rahaman et al., 2019), and an industry-based tool, Snyk Code (Snyk, nd). To evaluate LLM accuracy, we used well-known benchmark datasets for cryptographic misuse detection, namely OWASP Bench (OWASP Benchmark, 2016) and CryptoAPI Bench (Afrose et al., 2019). Additionally, we tested LLM robustness in detecting mutated test cases, where many static tools often struggle (Ami et al., 2022), and evaluated the quality of LLM responses using two metrics, Actionability and Specificity, to assess whether responses can help developers identify and fix misuses.

Our research aims to address the following research questions (RQs):

RQ1. How effective are LLMs in detecting cryptographic misuses compared to other static tools?

The static tools tested focus on slightly different pattern sets, leading to varied trade-offs in precision and recall, both among the tools themselves and in comparison to LLMs. To evaluate the effectiveness of LLMs in detecting cryptographic misuses compared to static tools, we ran test cases from the CryptoAPI and OWASP benchmarks. Based on the findings, GPT had a better detection rate across both benchmarks. For CryptoAPI, GPT missed only 3 true instances, with CryptoGuard lagging behind at 24 misses. In the OWASP benchmark, GPT identified all true instances, while CryptoGuard missed 40 true misuse cases. However, for the OWASP benchmark, GPT had a high false positive rate, indicating that no tool is universally superior across both benchmarks.

RQ2. How robust are LLMs in detecting mutated test cases that other static tools fail to detect?

To evaluate the robustness of LLMs, we ran test cases from a manually curated MASC dataset (Ami et al., 2022) against LLMs to see if LLMs can detect these mutations of cryptographic code effectively. Our results suggest that GPT performed better than other LLMs in detecting cryptographic mutations.

RQ3. How do prevalent LLMs compare in detecting cryptographic misuse and providing actionable, specific guidance for developers?

To address this, we compared the performance of LLMs across both benchmarks and a mutated dataset to evaluate which LLM performs best. Additionally, we introduced a method to assess whether an LLM's response can assist developers in fixing misuses. Since LLMs often generate text-heavy responses that may not effectively help developers, we implemented a keyword-based approach to analyze LLM outputs. This approach identifies which LLM provides more actionable and specific guidance for developers to address misuse instances.

Our main contributions from this work are as follows:

- We are the first to conduct a comprehensive evaluation of the effectiveness of LLMs in detecting cryptographic misuses, comparing their performance to that of SOTA static analysis tools. It highlights the strengths and weaknesses of LLMs compared to static tools, offering insights into their reliability and applicability for secure software development.
- We assess the robustness of LLMs by evaluating their ability to detect cryptographic misuse in mutated test cases that static tools often miss. This contribution explores whether LLMs can adapt to variations in misuse patterns, providing an understanding of their resilience and potential superiority in handling unconventional or complex misuse scenarios.
- To systematically assess the practicality of LLM-generated guidance, we develop a keyword-based framework that scans LLM responses for actionable elements. This framework serves as a tool to measure the usability of LLM outputs for developers, supporting the identification of models that best fulfill developers' needs in addressing cryptographic misuse. To the best of our knowledge,

this is the first work that explores the actionability and specificity of LLM responses.

In the next section, we cover the motivation for our work. Section 3 discusses background and related work. Section 4 explains the methodology, while Section 5 presents the results and findings. Section 6 covers the discussion, followed by Section 7 on ecological validity. Section 8 mentions the limitations and future directions, and finally we conclude our paper in Section 9.

2 MOTIVATION

To proactively identify vulnerabilities before software release, individuals without specialized expertise—such as software developers or quality assurance teams—are likely to integrate cryptographic API misuse detectors, commonly known as crypto-detectors, into the Continuous Integration/Continuous Delivery (CI/CD) pipeline. Examples of these detectors include CryptoGuard (Rahaman et al., 2019) and CogniCrypt (Krüger et al., 2017). Consequently, the incapability of a crypto detector to identify an instance of misuse that it claims to detect has a direct impact on the security of end-user software. We illustrate this challenge with an example.

Imagine Johnny, a Java developer utilizing CryptoGuard [8], an advanced crypto detector, to pinpoint cryptographic vulnerabilities in his software before its release. In one of his applications, Johnny opts to employ the DES cipher, outlined as follows:

Cipher cipher = Cipher . getInstance ("des");

This signifies an instance of misuse involving the use of the vulnerable DES cipher. CryptoGuard falls short in recognizing this vulnerability because Johnny utilizes "des" instead of "DES" as the parameter (Ami et al., 2022). However, this presents a concern because the difference in lower case for the parameter lacks functional significance, as Java officially supports both upper case and lower case. Due to CryptoGuard's oversight of this vulnerability, Johnny might erroneously believe that his application is secure and may proceed with releasing it to end-users.

The same example was tested with ChatGPT by providing a starting prompt "Can you detect cryptographic API misuse in the given Java code?". ChatGPT generated the following response:

"The line of code you provided initializes a Cipher object in Java using the DES (Data Encryption Standard) algorithm. However, it's important to note that DES is considered insecure for many applications due to its small key size (56 bits) and vulnerability to brute-force attacks. It is generally recommended to use more secure algorithms like AES (Advanced Encryption Standard) instead."

In other words, these static tools for cryptographic misuse detection could have fundamental flaws preventing them from detecting even straightforward instances of crypto-API misuse found in applications. On the other hand, ChatGPT's accurate response to such test cases raises a curiosity to test whether LLMs can effectively detect cryptographic misuses better than these static analysis tools. To extend it, novice developers intrigued by the power of LLMs capabilities, have started adapting to the technology by using such platforms to look for coding solutions. With the increasing use of AI to look for code solutions by novice developers, little research is done in this area to explore whether prominent platforms, such as ChatGPT, are effective in detecting these cryptographic misuses. It will also likely conclude whether the code provided by these LLMs for cryptographic problems is secure or not. This insight guides our approach to systematically compare widely known crypto detectors with LLMs.

Although early analysis showed promising results for Open AI's ChatGPT, an intensive study was conducted to conclude the effectiveness of LLMs in detecting cryptographic misuses. LLMs can generate and understand code in various programming languages, however, static tools and benchmarking datasets used for comparison in our study rely only on Java. Therefore, the scope of our work is limited to Java as a programming language.

3 BACKGROUND

Recently, security researchers have expressed significant interest in externally validating static analysis tools (Zhang et al., 2023; Ami et al., 2022). Specifically, there's a growing recognition that while static analysis security tools are theoretically sound, they can be "soundy" in practice. This means they consist of a core set of sound decisions but also include certain strategically unsound choices made for practical reasons, such as performance or precision considerations (CryptoGuardOSS, 2020). In this section, we provide a brief overview of the static analysis tools that will be used for comparison with LLMs. Moreover, we also discuss similar work done in this domain so far.

CryptoGuard (Rahaman et al., 2019) expands on

Table 1: Overview of Existing Work on LLM-Based Detection.

Name	Benchmarks			LLM-based Detection	Evaluating Quality of LLM Response
	CryptoAPI	MASC	OWASP		
Firouzi et al. (Firouzi et al., 2024)	✓	×	×	✓	×
Xia et al. (Xia et al., 2024)	✓	✓	×	✓	×
Our Work	✓	✓	✓	✓	✓

Soot (Soot, 2020), a widely used program analysis framework for statically analyzing Java bytecode (Vallée-Rai et al., 1999). It targets vulnerabilities associated with CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757, using context- and field-sensitive backward and forward slicing for precise detection. However, since eight of its rules involve constant value usage, traditional slicing techniques may falsely flag constants unrelated to security. To mitigate this, CryptoGuard integrates refinement algorithms that reduce false positives using cryptographic domain knowledge.

CogniCrypt (Krüger et al., 2017) employs rules specified in a domain-specific language (DSL) called CrySL (Krüger et al., 2021) to identify API misuses. CogniCrypt translates CrySL rules into context-sensitive, flow-sensitive, and demand-driven static analysis, enabling users to enhance the tool's capabilities by creating new rules. In case of identified API misuse, CogniCrypt provides guidance on its resolution, such as substituting an insecure parameter value with a secure one. Its pattern set pertains to CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757.

Snyk Code (Snyk, nd) is a static application security testing (SAST) tool that helps developers identify and fix code vulnerabilities. It integrates with development environments, continuously scanning codebases and providing actionable remediation insights. Notable for its developer-friendly approach, Snyk Code supports popular IDEs, CI/CD pipelines, and repositories. It uses machine learning and semantic analysis for accurate and relevant issue detection and supports multiple programming languages and frameworks, ensuring versatility.

Static analysis tools often suffer from a high rate of false positives, flagging issues that do not pose a threat and creating a disconnect between reported misuse alerts and real vulnerabilities. Chen et al. (Chen et al., 2024) examine these limitations by analyzing the rules, models, and implementations of such tools, highlighting the need for improvements in their precision and usability. In contrast, LLMs are becoming popular for code analysis. Fang et al. (Fang et al., 2024) found that advanced LLMs like ChatGPT 3.5 and 4.0 show promise in accurate code review. This indicates that LLMs could reduce false positives common in static tools, offering a more reliable alternative for vulnerability detection. Liu et al.(Liu et al., 2024) highlight ChatGPT's potential in vulnerability management, demonstrating its effectiveness in tasks like generating software bug report titles. As LLMs gain popularity, researchers are exploring their use in detecting cryptographic misuses(Firouzi et al., 2024; Xia et al., 2024). However, no prior research has investigated how LLMs' text-based responses help developers fix misuse instances.

Despite advances in static analysis tools for detecting cryptographic misuse, gaps remain in their coverage and adaptability to varied misuse patterns. Existing tools struggle with mutated misuse instances, reducing their effectiveness in real-world scenarios. Prior research (Masood and Martin, 2024) has explored LLMs for detecting cryptographic misuses but has been limited in model comparison. This study extends previous work by including a broader range of LLMs for a more thorough evaluation against static tools and among LLMs. It introduces ROC-annotated plots for clearer performance comparisons. Our study evaluates static tools and LLMs using OWASP and CryptoAPI benchmarks, testing LLMs' robustness with mutated misuse cases. We also introduce a keyword-based approach to assess the actionability and specificity of LLM responses in aiding code correction. By addressing these aspects, our work explores the potential of replacing static tools with LLMs to help developers write secure code. Table 1 highlights the unique contributions of our work compared to prior research.

4 METHODOLOGY

This section describes our methodology for evaluating LLMs in cryptographic misuse detection. We assessed LLMs against various datasets and further analyzed their responses to determine their effectiveness in detecting misuse instances.

4.1 Selection Criteria

4.1.1 Static Tools

A range of tools exist for detecting cryptographic vulnerabilities in Java code, with most prior research focusing on this language. We selected two academic tools, CryptoGuard (Rahaman et al., 2019) and CogniCrypt SAST (Krüger et al., 2017), and one industry tool, Snyk Code (Snyk, nd). We prioritized tools that accept JAR files and are generic, excluding Android-specific tools. We also favored widely recognized tools to enhance the credibility of our comparison with LLM models. CryptoGuard and CogniCrypt were set up on an Ubuntu VM, with CryptoGuard requiring Java 8 and CogniCrypt requiring Java 11 or higher. Snyk Code was installed as a Visual Studio Code extension. Table 2 lists the static analysis tools and their versions. Results from all tools were extracted as JSON files and analyzed against benchmark labels.

Table 2: Version of Static Analysis Tools.

Tool	Version
CryptoGuard	04.05.03
CogniCrypt	3.0.2 and 4.0.1
Snyk Code	2.18.2

4.1.2 Benchmark Datasets

To evaluate the tools' effectiveness, we conducted extensive online searches for open-source benchmarks that classify programs based on correct or incorrect use of cryptographic APIs. We selected three widely used datasets: CryptoAPI Bench (Afrose et al., 2019), OWASP Benchmark (OWASP, n.d.), and MASC dataset (Ami et al., 2022). These datasets provide test cases for assessing cryptographic misuses in Java programs.

- The CryptoAPI Bench (Afrose et al., 2019) consists of 182 test cases, with 181 labeled and one unlabeled. Of the labeled cases, 144 are true misuses, and 37 are false misuses. The unlabeled case was excluded from testing.
- The OWASP Benchmark (OWASP, n.d.) is a Java test suite evaluating vulnerability detection methods, incorporating vulnerabilities from the Common Weakness Enumeration (CWE - Common Weakness Enumeration, 2021). Version 1.2 contains 2,740 Java programs, but the study focuses on 975 programs categorized into weak cryptography, weak hashing, and weak randomness, with 477 showing misuse and 498 showing proper use.
- The MASC dataset, designed by Ami et al. (Ami et al., 2022), applies mutation testing to identify vulnerabilities in crypto detectors, assessing their ability to withstand code modifications. We manually selected 30 test cases from MASC's minimal test suites, targeting five prevalent flaw types in modern crypto detectors.

The test case outline for the benchmarking datasets is shown in Table 3. We chose existing benchmark datasets for two reasons: they are publicly accessible, curated by diverse stakeholders, ensuring a comprehensive and replicable empirical comparison, and benchmarks like OWASP Benchmark are widely recognized and influential in the industry. Using these datasets with static analysis tools and LLMs helped answer RQ1, while testing LLMs with mutations addressed RQ2.

Table 3: Labels of Benchmarking Datasets.

Benchmarking Dataset	True Labels	False Labels	Total
CryptoAPI Bench	144	37	181
MASC	29	1	30
OWASP	477	498	975

4.1.3 Large Language Models

The popular LLM models selected for this study include OpenAI's GPT, Google's Gemini, Meta's Llama, and Anthropic's Claude, chosen for their advanced capabilities in converting text to code and identifying vulnerabilities through code analysis. Each misuse instance was provided with a standardized prompt (details in Appendix), which included information like the method name, starting line number, highlighted message, message description (reason for misuse), lines of code with the misuse, and a label indicating cryptographic misuse. The LLM responses followed a similar pattern, providing the misuse label, method name, starting line number, code line, highlighted message, cause explanation, and best practices. The LLM models and their versions are listed in Table 4.

Table 4: Version of Large Language Models.

LLM	Version	Release Date
Meta's Llama	Llama 3.0	April 18, 2024
Open AI's GPT	gpt-4o-mini-2024-07-18	July 18, 2024
Google's Gemini	gemini-1.5-flash-002	September 24, 2024
Anthropic's Claude	Claude 3.5 Haiku	October 22, 2024

4.2 Evaluation Metrics

Accuracy of each static tool and LLMs will be measured by these 8 metrics: ***True Positive, False***

Positive, False Negative, True Negative, Precision, Recall, F1-score, and F2-score.

While the F1-score is an effective measure for evaluating a tool's performance, it should not be the only criterion for tool selection. In a security context, placing more emphasis on recall with F2-score will be more reliable than relying solely on the F1-score. For this reason, our study uses the F2-score alongside the F1-score.

4.2.1 LLM Specific Metrics

Since Large Language Models generate text, traditional performance metrics such as precision, recall, and F-score are insufficient for fully assessing their capabilities. Therefore, in addition to these metrics, we employed two measures influenced by Redmiles et al. (Redmiles et al., 2020) to evaluate security and privacy advice: ***Actionability*** and ***Specificity***.
To evaluate these metrics, we utilized specific keywords proposed by Rostami et al. (Rostami and Karlsson, 2024) to deduce actionability in information security policies. We utilized their proposed list of keywords to evaluate whether the response from LLM is actionable. The keywords are listed in Table 5. For instance, if the text contains words such as "should", "need", or "must", we categorize it as "Actionable". Furthermore, a response is considered higher "Specificity" if it includes a link to a resource (e.g., a URL) or CWE.

Table 5: Actionable Keywords.

Keywords
never
need
should
not
forbidden
must
shall

For example, one of the mutated test cases replaces AES with DES by replacing "A" with "D". The misuse line is as follows:

```
Cipher c = Cipher.getInstance(''AES''.
replace(''A'', ''D''));
```

For this particular case, GPT gave the misuse description as:
"The usage of "AES".replace("A", "D")' is a significant misuse. This expression incorrectly modifies the string that is supposed to define the algorithm name, changing it from "AES" to "DES". Since "DES" is a less secure algorithm, this can lead to vulnerabilities in the encryption implementation realized by this code. This does ***not*** *represent a proper way to specify or select cryptographic algorithms, as it can easily be manipulated or misunderstood."*

The presence of an actionable keyword "not" in GPT's response provides clear feedback to developers to fix this misuse instance.

In contrast, the absence of actionable keywords in an LLM response may limit developers' ability to understand and address the misuse effectively. For instance, GPT provided the following response in another case where AES was replaced by DES:
"The issue arises because the program allows for the selection of weak ciphers (like DES) through method chaining without any safeguards or validation mechanisms to ensure that only strong ciphers are used. This directly impacts the security posture of the application by making it susceptible to various cryptographic attacks, thereby constituting a cryptographic misuse."

Here, the lack of actionable keywords may limit the guidance that novice developers need, reducing the response's effectiveness in directing them toward an appropriate fix and failing to emphasize the potential consequences of the misuse.

4.3 Experiments

Figure 1 shows our evaluation methodology for LLM-based cryptographic misuse detection. The methodology is divided into three sections: LLM vs Static Tools, Comparing LLMs, and Evaluating LLM responses.

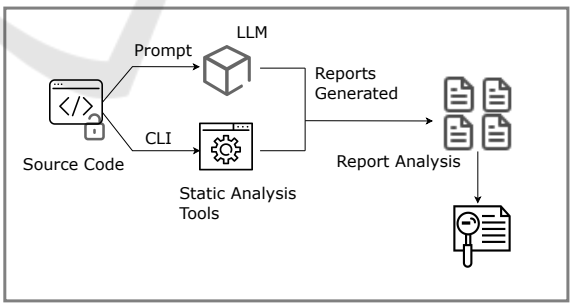

Figure 1: Evaluation Methodology for Comparing Static Tools with LLMs.

4.3.1 LLM vs Static Tools

We began our experiments by testing LLMs and static tools with the CryptoAPI and OWASP benchmarks to evaluate their effectiveness in detecting cryptographic misuses, using the evaluation metrics outlined in the metrics section. This helped us understand how well each approach identifies cryptographic misuses and

allowed us to compare the strengths of LLMs with those of static tools. This contributed to our findings on RQ1.

4.3.2 Comparing LLMs

In the second phase of our experiments, we tested LLMs using mutated test cases from the MASC dataset, along with CryptoAPI and OWASP benchmarks, to evaluate their effectiveness in detecting cryptographic misuses. This enabled us to assess the robustness of these LLMs in identifying cryptographic misuse across varied and altered scenarios, concluding our findings for RQ2.

4.3.3 Evaluating LLM Responses

Additionally, we analyzed the responses of LLMs using the Actionability and Specificity metrics. If a response from an LLM contained a keyword from Table 5, it was considered actionable. The number of actionable responses in a particular dataset represents the total actionability of the LLM for that dataset. To determine the Specificity of an LLM response, we manually reviewed each response to check if the LLM provided a link, referenced a CWE, or included recommended code that could assist developers in fixing the misuse. If the response from an LLM contained a link, CWE reference, or fixed code, it was considered specific. The number of test cases with specific responses in a particular dataset indicates the specificity of the LLM and is compared with other LLMs to assess specificity.

This comprehensive approach offered a clearer understanding of the responses provided by LLMs, assessing whether each response qualified as actionable advice for developers. Additionally, it revealed whether any links provided by the LLMs effectively guide developers to relevant resources for addressing specific cryptographic misuses, thus contributing insights for RQ3.

5 RESULTS AND FINDINGS

In this section, we present the results of our work on detecting cryptographic misuses for CryptoAPI and OWASP benchmarks, gathered using the selected static tools and GPT. Furthermore, we extended the results by comparing LLMs on the MASC dataset to determine which LLM performed better. We also analyzed the responses from LLMs to gain insights into which one offered more actionable and specific advice.

5.1 LLMs vs Static Tools

5.1.1 CryptoAPI Benchmark

For CryptoAPI benchmark, GPT 4-o-mini achieved the highest F1-score of 87.6%, F2-score of 93.5%, and a recall of 97.9%, accurately detecting 141 out of 144 misuse cases. CryptoGuard followed with an F1-score of 84.8%, F2-score of 83.9%, and a recall of 83.3%, missing 24 misuse cases. Detailed metrics are shown in Table 6.

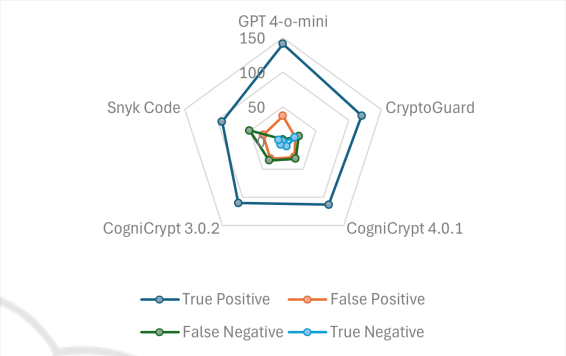

Figure 2: Tools Comparison on CryptoAPI Benchmark.

GPT 4-o-mini's high accuracy suggests its effectiveness as an alternative to traditional static analysis tools for cryptographic misuse detection. Unlike static tools that depend on fixed rules, GPT 4-o-mini utilizes advanced pattern recognition and contextual adaptability, enabling it to handle complex misuse cases with high precision and recall, although it shows a slightly higher rate of false positives.

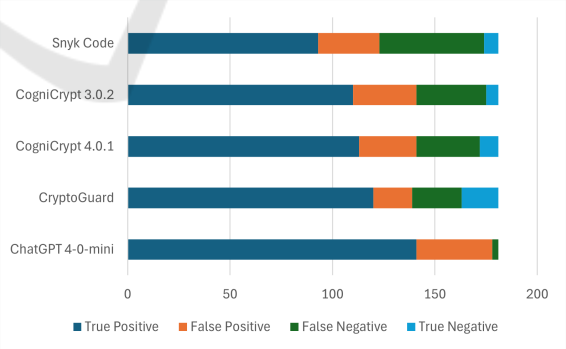

Figure 3: Detection Results for CryptoAPI Benchmark Across Different Tools.

This adaptability allows GPT 4-o-mini to outperform static tools like CryptoGuard and CogniCrypt, which are constrained by predefined rules and may miss specific misuse types, such as Java's Secret Key Factory or Hostname Verifier API cases. Key metrics are illustrated in Figure 2 and Figure 3.

Table 6: Evaluation Metrics for CryptoAPI and OWASP Benchmark.

Benchmark	Tool	True Positive	False Positive	False Negative	True Negative	Precision (%)	Recall (%)	F1 Score (%)	F2 Score (%)
CryptoAPI	GPT 4-o-mini	**141**	37	**3**	0	79.2	**97.9**	**87.6**	**93.5**
	CryptoGuard	120	**19**	24	**18**	**86.3**	83.3	84.8	83.9
	CogniCrypt 4.0.1	113	28	31	9	80.1	78.5	79.3	78.8
	CogniCrypt 3.0.2	110	31	34	6	78.0	76.4	77.2	76.7
	Snyk Code	93	30	51	7	75.6	64.6	69.7	66.5
OWASP	CryptoGuard	437	**27**	40	**471**	**94.2**	91.6	**92.9**	**92.1**
	GPT 4-o-mini	**477**	498	**0**	0	48.9	**100.0**	65.7	82.7
	Snyk Code	477	498	**0**	0	48.9	**100.0**	65.7	82.7
	CogniCrypt 4.0.1	259	200	218	298	56.4	54.3	55.3	54.7
	CogniCrypt 3.0.2	259	475	218	23	35.3	54.3	42.8	49.0

Finding 1 RQ1: GPT 4-o-mini achieved the highest F-score (F1-score: 87.6%, F2-score: 93.5%) among tools for CryptoAPI benchmark, showing strong cryptographic misuse detection despite not being specialized, followed closely by CryptoGuard. Snyk Code and CogniCrypt had higher error rates, making them less reliable overall.

5.1.2 OWASP Benchmark

For OWASP benchmark, CryptoGuard achieved the highest F1-score of 92.9%, F2-score of 92.1%, with a precision of 94.2% and a recall of 91.6%, effectively detecting true misuse cases while minimizing false positives. Snyk Code and GPT 4-o-mini followed with an F1-score of 65.7% and F2-score of 82.7%, where the higher F2-score reflects stronger emphasis on recall, highlighting GPT 4-o-mini's ability to catch more misuse cases despite lower precision. Results are summarized in Table 6.

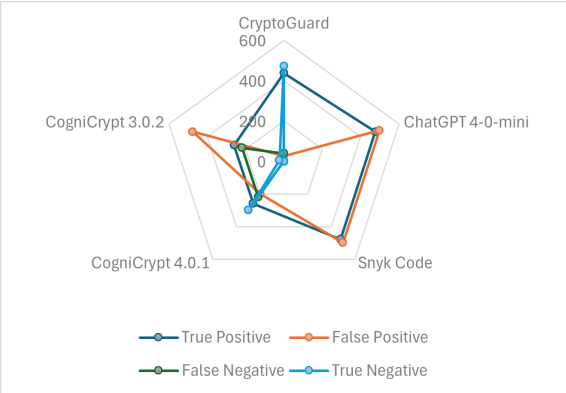

Figure 4: Tools Comparison on OWASP Benchmark.

GPT 4-o-mini's high false positive rate stems from its text-based responses, which frequently suggest best practices even when no misuse is present. This led to labeling cases as "TRUE" due to best practice recommendations, even without actual misuse, reducing its precision in identifying true misuse cases. Additionally, the OWASP test cases had a higher average line count compared to CryptoAPI test cases, increasing the amount of code GPT had to analyze. With longer code samples, GPT often provided general recommendations rather than focusing solely on detecting cryptographic misuses, leading to more false positives. However, GPT performed more accurately with smaller code samples than with longer ones. Figure 4 and Figure 5 illustrate the key metrics for tools tested on the OWASP Benchmark, showcasing CryptoGuard's superior performance compared to other tools.

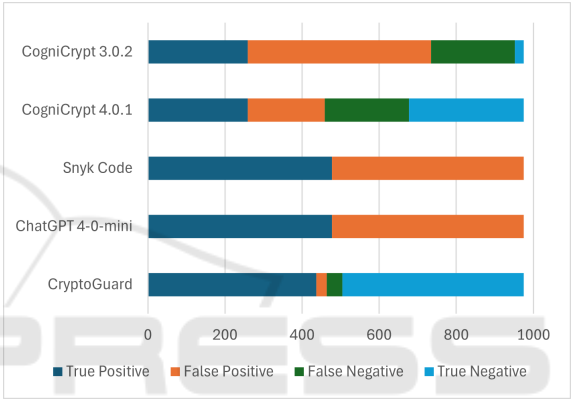

Figure 5: Detection Results for OWASP Benchmark Across Different Tools.

Finding 2 RQ1: CryptoGuard outperforms other tools on the OWASP benchmark with a 92.1% F2-score, while GPT and Snyk Code lag behind at 82.7%, demonstrating CryptoGuard's higher accuracy in detecting cryptographic misuses.

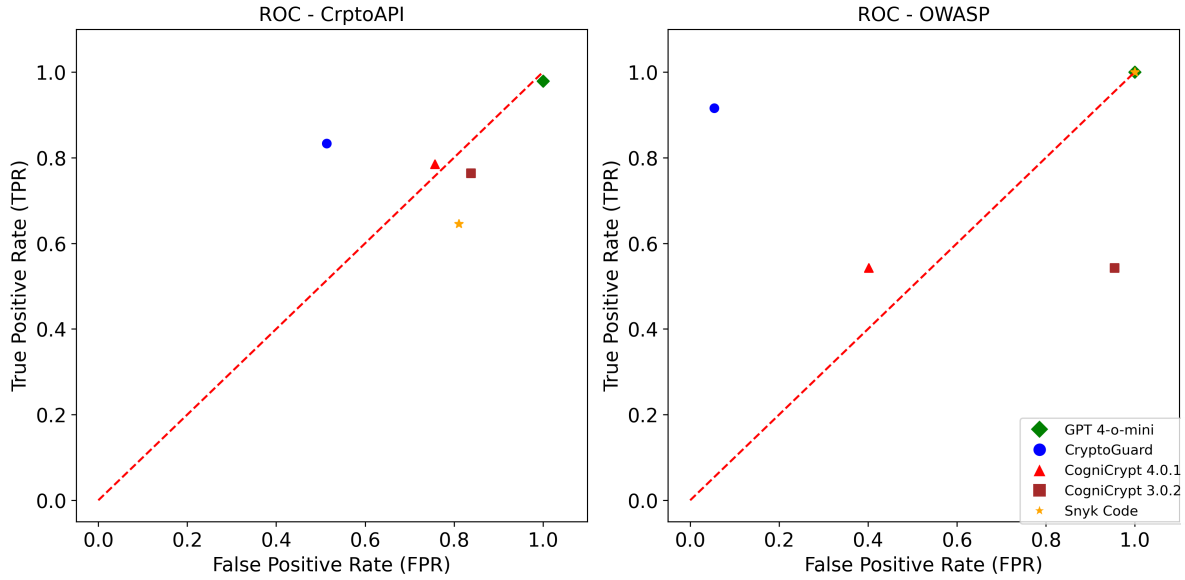

Figure 6: TPR vs FPR for Static Tools and GPT.

5.1.3 ROC Comparison

Receiver Operating Characteristic (ROC) curves help visualize and compare the trade-off between true and false positives across different detection thresholds. It provides an intuitive way to assess how well each model distinguishes between correct and incorrect cryptographic API usage. The comparison of static tools with GPT across the CryptoAPI and OWASP benchmarks revealed varied performance. CryptoGuard effectively balanced true positive detection and false positive reduction, outperforming other tools, as shown in Figure 6. GPT 4-o-mini had high recall but also a high false positive rate, indicating challenges with precision in LLM-based vulnerability detection. CogniCrypt and Snyk Code performed similarly to random guessing on CryptoAPI. CogniCrypt struggled with diverse misuse instances, leading to higher false positives. The CryptoAPI results were affected by an imbalance in misuse instances. Reducing false positives is crucial for practical use, and future research should include probability mechanisms for more accurate comparisons.

Finding 3 RQ1: CryptoGuard demonstrates a good balance between detecting true positives and minimizing false positives, outperforming other tools and GPT in some cases.

5.1.4 Time Comparison

The benchmarks were run on tools configured in an Ubuntu VM hosted on a laptop with the following specs: Windows 11 OS, i7-1255U CPU (1.70 GHz), 16 GB RAM. The VMs used for the tools had 10 GB RAM, an 8 GB JVM heap, and 4 processors.

Test cases were executed on GPT via an API, and response times for each benchmark were recorded. Static analysis tools, including CryptoGuard, CogniCrypt, and Snyk Code, were run using CLI commands and Visual Studio Code extension. The time taken by each tool for both benchmarks is shown in Table 7.

Table 7: Execution Time of Tools for CryptoAPI and OWASP Benchmark.

Tools	CryptoAPI	OWASP
CryptoGuard	1m 48s	10m 59s
CogniCrypt 3.0.2	54s	16m 35s
CogniCrypt 4.0.1	1m 7s	23m 26s
Snyk Code	**8s**	**52s**
GPT 4-o-mini	8m 45s	71m

Snyk Code was the fastest tool across both benchmarks, with execution times of 8 seconds for CryptoAPI and 52 seconds for OWASP. CryptoGuard performed well on the CryptoAPI benchmark (1m 48s) but took longer on the OWASP benchmark (10m 59s). GPT 4-o-mini had the longest execution times, requiring 8m 45s for CryptoAPI and 71 minutes for OWASP, highlighting a trade-off between detection accuracy and slower processing speed, which could limit scalability for large datasets.

Finding 4 RQ1: Snyk Code is the fastest tool for both benchmarks, followed by CryptoGuard. GPT 4-o-mini, while strong in detection, has slower processing speeds, limiting its scalability for larger datasets.

5.2 Comparing LLMs with Datasets

The performance of selected LLMs was evaluated on benchmark datasets, including mutated test cases, to identify the best model. GPT 4-o-mini was accessed via API, while Gemini 1.5 Flash, Llama 3.0, and Claude Haiku 3.5 were accessed via chat interfaces. The models were tested on 30 random test cases from the CryptoAPI and OWASP benchmarks, as well as 30 manually curated MASC test cases.

For the CryptoAPI benchmark, Claude achieved the highest F2-score of 94.3% and recall of 100%, followed by GPT (97.9% recall, 93.5% F2-score), and Gemini with the lowest F2-score of 85.5%. For the OWASP benchmark, GPT led with an F2-score of 82.7%, followed by Gemini at 81.1%, while Llama and Claude had the lowest F2-scores, both at 76.9%. All LLMs demonstrated perfect recall (100%), with Claude showing better performance for the CryptoAPI benchmark and GPT performing better for the OWASP benchmark, as detailed in Table 8.

On the mutation dataset (MASC), GPT and Claude both achieved 100% recall and an F2-score of 99.3%, detecting all 29 true misuse cases. Gemini, however, had the lowest performance with a recall of 79.3% and an F2-score of 82.7%, identifying 23 out of 29 misuse cases. These results emphasize the effectiveness of LLMs in scenarios where static tools face challenges.

Finding 1 RQ2: Claude and GPT achieved comparable F2-scores of 99.3% on the MASC benchmark, performing better than other LLMs and indicating similar effectiveness in detecting cryptographic misuses. GPT outperformed all models on the OWASP benchmark with an F2-score of 82.7%, while Claude delivered the best results on the CryptoAPI benchmark with an F2-score of 94.3%.

5.2.1 ROC Comparison

Among LLMs, Gemini 1.5 Flash performs better than the others, correctly categorizing false instances as false and falling to the left of the random guessing line (Figure 7 to Figure 9).

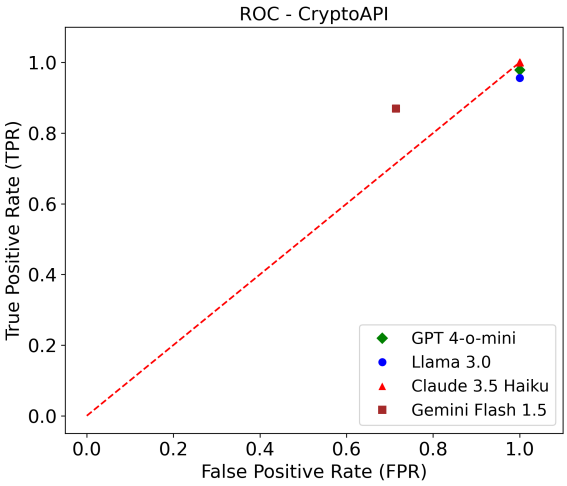

Figure 7: TPR vs FPR of LLMs for CryptoAPI.

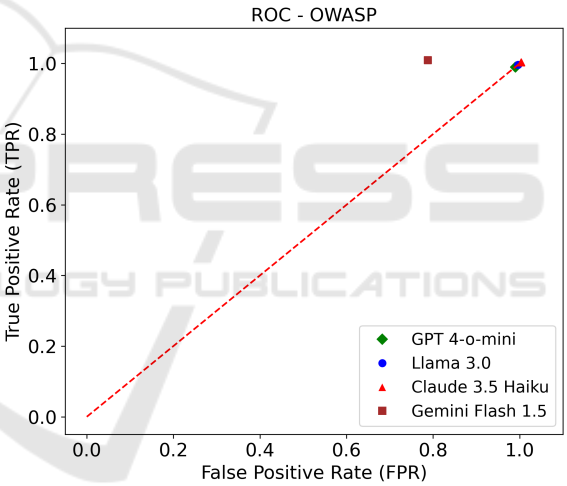

Figure 8: TPR vs FPR of LLMs for OWASP.

In contrast, GPT-4-o-mini, Llama 3.0, and Claude 3.5 Haiku are closer to the random guessing line, correctly identifying true instances but misclassifying false ones. These results are influenced by class imbalance in MASC, where true instances dominate, and only a single false instance is present. Additionally, the results are affected by the fact that, apart from GPT, all other LLMs were tested against only a subset of misuse instances from the CryptoAPI and OWASP benchmarks. While LLMs show potential, reducing false positives and addressing class imbalance remain crucial for ensuring a fair comparison.

[1]Gemini, Llama, and Claude were evaluated against random 30 test cases for CryptoAPI and OWASP datasets.

Table 8: Comparison of LLMs on Different Benchmarks.[1]

Benchmark	LLM	True Positive	False Positive	False Negative	True Negative	Precision (%)	Recall (%)	F1 Score (%)	F2 Score (%)
CryptoAPI	GPT 4-o-mini	**141**	37	3	0	79.2	97.9	**87.6**	93.5
	Llama 3.0	22	7	1	0	75.9	95.7	84.6	91.0
	Claude 3.5 Haiku	23	7	**0**	0	76.7	**100**	86.8	**94.3**
	Gemini Flash 1.5	20	**5**	3	**2**	**80.0**	87	83.3	85.5
MASC	GPT 4-o-mini	**29**	1	**0**	0	96.7	**100**	**98.3**	**99.3**
	Llama 3.0	28	1	1	0	96.6	96.6	96.6	96.6
	Claude 3.5 Haiku	29	1	**0**	0	96.7	**100**	**98.3**	**99.3**
	Gemini Flash 1.5	23	**0**	6	**1**	**100**	79.3	88.5	82.7
OWASP	GPT 4-o-mini	**477**	498	**0**	0	**48.9**	**100**	**65.7**	**82.7**
	Llama 3.0	**12**	18	**0**	0	40	**100**	57.1	76.9
	Claude 3.5 Haiku	**12**	18	**0**	0	40	**100**	57.1	76.9
	Gemini Flash 1.5	**12**	**14**	**0**	**4**	46.2	**100**	63.2	81.1

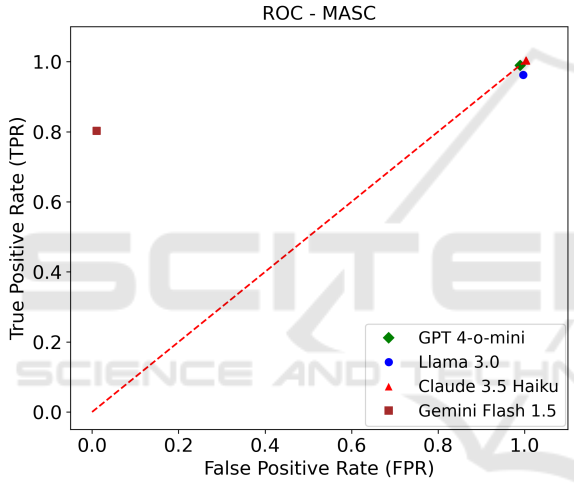

Figure 9: TPR vs FPR of LLMs for MASC.

Finding 2 RQ2: Gemini strikes a good balance between detecting true positives and minimizing false positives, although GPT and other LLMs outperform it with higher recall and F-scores.

5.3 Evaluating LLM Responses

Actionability and Specificity were used to evaluate LLM responses. Actionability was determined by checking for actionable keywords, while Specificity was assessed by the inclusion of CWE references, external resources, or fixed code. The percentage of actionable and specific responses was calculated for each benchmark to compare LLMs. The results for actionability and specificity across benchmarks are presented in Table 9.

GPT 4-o-mini provided the most actionable responses on MASC (63.3%) and CryptoAPI (61.4%) but had 0% specificity across all benchmarks, indicating a lack of detailed guidance. Llama 3.0 achieved the highest actionability on OWASP (90.0%) but also showed no specificity. In contrast, Claude 3.5 Haiku excelled in specificity, scoring 43.3% on OWASP and 46.7% on MASC, while maintaining high actionability on OWASP (86.7%) and moderate levels on CryptoAPI (53.3%). Overall, GPT 4-o-mini demonstrated strong actionability but lacked detailed guidance. Llama 3.0 performed best in actionability for OWASP, while Claude 3.5 Haiku balanced actionability with the highest specificity by providing CWE references, external links, or fixable code for cryptographic misuses.

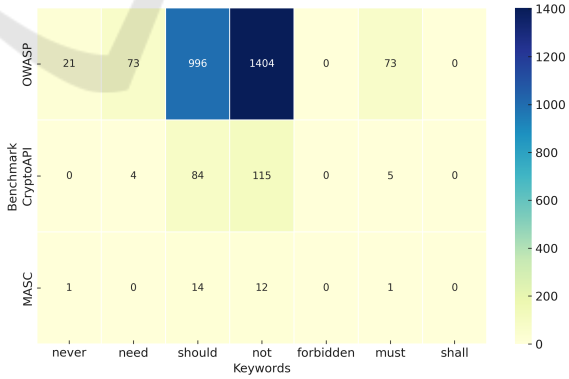

Figure 10: GPT Actionable Keywords Occurence.

Figures 10 to 13 shows the occurrence of actionable keywords for the selected LLMs. The most frequently used keywords across benchmarks are “not” and “should”. In both the OWASP and CryptoAPI benchmarks, “should” is common, with 996 instances in GPT for OWASP, and a lower but consistent pres-

Table 9: Actionability and Specificity of LLMs Across Benchmark Datasets.

Benchmark	LLM	Actionable Responses	Specific Responses	Total Responses	Actionability (%)	Specificity (%)
CryptoAPI	GPT 4-o-mini	162	0	264	**61.4**	0.0
	Llama 3.0	14	2	30	46.7	**6.7**
	Claude Haiku 3.5	16	2	30	53.3	**6.7**
	Gemini Flash 1.5	9	2	30	30.0	**6.7**
MASC	GPT 4-o-mini	19	0	30	**63.3**	0.0
	Llama 3.0	15	2	30	50.0	6.7
	Claude Haiku 3.5	15	14	30	50.0	**46.7**
	Gemini Flash 1.5	7	7	30	23.3	23.3
OWASP	GPT 4-o-mini	1584	0	1909	83.0	0.0
	Llama 3.0	27	0	30	**90.0**	0.0
	Claude Haiku 3.5	26	13	30	86.7	**43.3**
	Gemini Flash 1.5	21	6	30	70.0	20.0

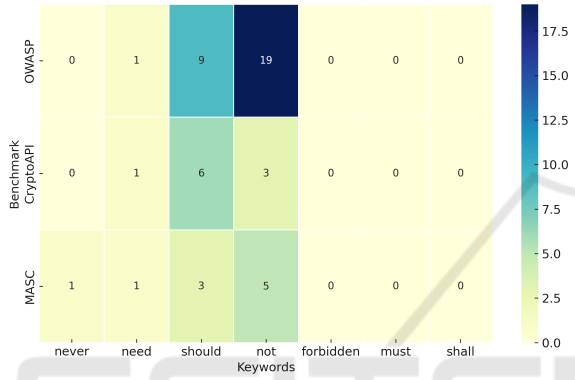

Figure 11: Gemini Actionable Keywords Occurence.

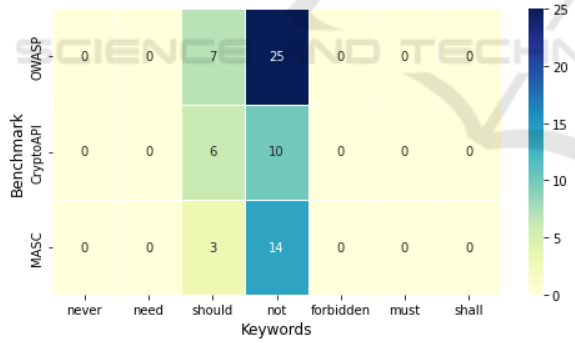

Figure 12: Llama Actionable Keywords Occurence.

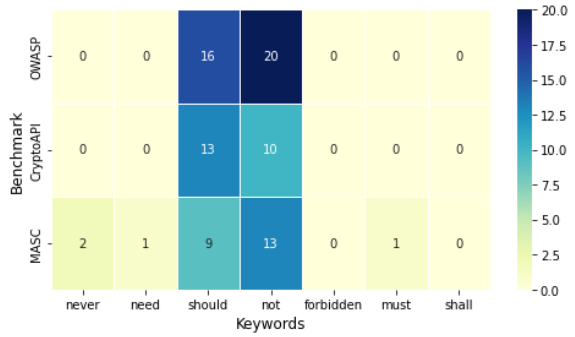

Figure 13: Claude Actionable Keywords Occurence.

ence across other benchmarks, providing positive actionable feedback for developers. Similarly, "not" appears frequently, especially in OWASP, with 1404 instances in GPT, providing negative actionability and guiding developers on cryptographic misuses. "Forbidden" and "shall" were the least used keywords across different benchmarks for the selected LLMs, highlighting a focus on suggestion and negation, guiding actions to consider or avoid. Less frequent keywords, such as "need", "must", "never" and "shall", suggest a preference for softer language rather than strict requirements.

Finding RQ3: GPT 4-o-mini excels in actionability across multiple benchmarks, providing actionable feedback on things to consider and avoid, but lacks specificity. In contrast, while Claude 3.5 Haiku exhibits lower actionability, it provides greater specificity compared to other LLMs.

6 DISCUSSION

Strengths of Large Language Models. LLMs have an inherent advantage over static analysis tools as static tools rely on a rule-based approach. Any deviations of misuse instance from a rule will likely cause the tool to miss it. However, LLM detection capability is not particular to a specific rule set, and it can detect misuse instances in a variety of code as highlighted by the results of mutated test cases. Similarly, LLMs are not specific to a certain programming language. It can read and understand various programming languages as opposed to static analysis tools that are specific to a programming language. LLMs do not have a framework dependency and now integrations are being supported to adopt LLMs for various frameworks which will likely increase its usage in software development. Even though LLMs currently have a high false positive rate for detecting

cryptographic misuses, this is likely to improve as they are trained on more data. Better accuracy than static analysis tools in detecting true alerts for security context shows LLMs can be utilized in this domain.

Benchmark Inconsistencies. Our analysis of the CryptoAPI benchmark revealed inconsistencies, including missing test cases in the dataset that were listed in the label sheet, such as CredentialInStringBBCase2.java and PredictableSeedsABPMCase2.java. This mismatch can lead to false negatives and misrepresent the tools' effectiveness. Additionally, some Java test cases lacked corresponding labels, causing tools to detect misuse without reference. To ensure consistency, we excluded these unlisted test cases, emphasizing the need for a comprehensive, standardized dataset to properly evaluate tool effectiveness across all Java API categories.

7 ECOLOGICAL VALIDITY

7.1 Applicability to Real World Scenarios

Our experiments utilized the OWASP Benchmark, a well-known open-source Java suite that serves as a standard for assessing application security. The CryptoAPI benchmark was also included based on a review of previous studies, where it has been extensively used. Additionally, the MASC dataset was selected to capture the diverse ways in which code can be written by developers. The results obtained from these benchmarks provide insights into the effectiveness of different tools and LLMs in detecting vulnerabilities across various domains. Our focus was specifically on cryptographic misuses, making the results highly relevant for comparing our tool with both industry-standard and academic tools, many of which are evaluated against these benchmarks. Since these benchmarks are commonly used, their results are a reliable measure of a tool's accuracy in identifying cryptographic misuses.

The increasing adoption of LLMs by novice developers, often in place of traditional search engines, reflects a shift in how coding solutions are sourced. Many coding environments, such as Visual Studio Code, now integrate LLM-based assistants like ChatGPT, allowing developers to prompt these models directly. With LLMs offering quick, context-specific answers, novice developers increasingly rely on these tools not only for general coding guidance but also for secure coding practices, including cryptographic implementations. As more developers turn to LLMs for solutions, they inherently depend on the models' security accuracy to prevent cryptographic misuses in their code. At the same time, integrating LLM-based detection into the CI/CD pipeline ensures that cryptographic issues are automatically flagged before deployment. This helps teams enforce secure coding practices continuously, reduces the risk of introducing security flaws into production code, and addresses cryptographic vulnerabilities as part of the development lifecycle with minimal manual effort from developers. Therefore, it becomes critical to evaluate how well LLMs address security contexts, ensuring that the solutions provided are both effective and safe. Understanding this trend highlights the real-world importance of our work, which assesses the quality and security of LLM responses in situations where developers may not be aware of potential misuses.

7.2 Threats to Validity

Threats to Internal Validity The scope of our findings may be limited by the specific tools and datasets chosen for testing. To address this, we plan to expand our evaluation by including additional tools and testing them on examples drawn from real-world cryptographic code. Another potential limitation is data leakage, as the open-source benchmarks used for testing might have been utilized in the training of LLMs, potentially affecting our results; however, the high number of false positives suggests otherwise. In the future, we plan to evaluate test cases by anonymizing class names to prevent any possible data leakage. LLMs may hallucinate non-existent vulnerabilities or incorrect patterns, leading to false positives that require manual verification. Additionally, the actionability and specificity metrics evaluated in our study may not fully translate into practical guidance for developers to fix code in every instance.

Threats to External Validity Our results may not generalize beyond the specific tools and datasets used in this study. To address this, we plan to expand our evaluation by incorporating additional tools and real-world cryptographic benchmarks. Additionally, our findings are specific to Java and may not apply in the same way to other programming languages. Moreover, we tested for mutations in test cases that were specific to Java as a programming language, the accuracy of detecting mutant misuses might not apply to other programming languages.

8 FUTURE WORK

Future research could explore testing LLMs on real-world codebase benchmarks, such as the Apache CryptoAPI benchmark, to assess performance on complex cryptographic scenarios. Expanding the study to include newer versions of LLMs, particularly those specialized in code generation, may provide insights into the benefits of using code-focused models over general-purpose ones. Additionally, testing LLMs on benchmarks in other programming languages could reveal language-specific strengths and limitations.

9 CONCLUSION

Cryptographic misuses in software pose significant security risks. Our study investigated how LLMs compare with traditional static analysis tools in detecting such vulnerabilities. We evaluated their performance across established benchmarks—CryptoAPI, OWASP, and MASC—and found that LLMs, particularly GPT, often achieved higher accuracy than static tools like CryptoGuard, especially on the CryptoAPI and MASC datasets. However, CryptoGuard outperformed LLMs on the OWASP dataset, highlighting that no single approach is universally superior.

While LLMs show strong potential, their relatively high false positive rate may hinder adoption by overwhelming developers with alerts. To address this, we introduced Actionability and Specificity as complementary metrics, offering deeper insight beyond precision and recall. Results showed that GPT offered more actionable suggestions, while Claude excelled in specificity—underscoring trade-offs in LLM-generated guidance. Integration of LLMs into CI/CD pipelines or development environments could enhance secure coding practices, but their effectiveness will rely on reducing false positives and improving contextual relevance. As these models evolve, they could complement existing tools and provide real-time, intelligent support for secure software development.

ACKNOWLEDGEMENTS

We thank the support of a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada (Grant # RGPIN/005919-2018) and the Vulnerability Research Centre at the Communications Security Establishment.

REFERENCES

Afrose, S., Rahaman, S., and Yao, D. (2019). CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *2019 IEEE Cybersecurity Development*, pages 49–61.

Amazon Q Developer (n.d.). Amazon Q Developer. https://aws.amazon.com/q/developer/. Retrieved August 21, 2024.

Ami, A. S., Cooper, N., et al. (2022). Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques. In *2022 IEEE Symposium on S&P*, pages 614–631.

ChatGPT (n.d.). ChatGPT. https://chat.openai.com/. Retrieved August 21, 2024.

Chatzikonstantinou, A., Ntantogian, C., et al. (2016). Evaluation of Cryptography Usage in Android Applications. In *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies*, BICT'15, page 83–90, Brussels, BEL. ICST.

Chen, Y., Liu, Y., et al. (2024). Towards Precise Reporting of Cryptographic Misuses. In *Proceedings 2024 NDSS*.

Claude (n.d.). Claude. https://claude.ai/. Retrieved December 21, 2024.

CryptoGuardOSS (2020). cryptoguardoss/cryptoguard. https://github.com/CryptoGuardOSS/cryptoguard. [Online; accessed August 21, 2024].

CWE - Common Weakness Enumeration (2021). CWE - Common Weakness Enumeration. https://cwe.mitre.org. [Online; accessed August 21, 2024].

Egele, M., Brumley, D., et al. (2013). An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 73–84, New York, NY, USA. Association for Computing Machinery.

Fang, C., Miao, N., et al. (2024). Large Language Models for Code Analysis: Do LLMs Really Do Their Job? In *33rd USENIX Security Symposium*, pages 829–846, Philadelphia, PA. USENIX Association.

Firouzi, E., Ghafari, M., et al. (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*, ESEM '24, page 582–588, New York, NY, USA. ACM.

Gemini (n.d.). Gemini. https://gemini.google.com/. Retrieved August 21, 2024.

GitHub AI Pair Programmer (n.d.). GitHub AI Pair Programmer. https://copilot.github.com. Retrieved August 26, 2024.

Graff, M. G. and Wyk, K. R. V. (2003). *Secure Coding: Principles and Practices*. O'Reilly & Associates, Inc., USA.

Kafader, S. and Ghafari, M. (2021). Fluentcrypto: Cryptography in easy mode. In *2021 ICSME*, pages 402–412.

Krüger, S., Nadi, S., et al. (2017). CogniCrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on ASE*, pages 931–936.

Krüger, S., Späth, J., et al. (2021). CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. *IEEE Transactions on Software Engineering*, 47(11):2382–2400.

Lazar, D., Chen, H., et al. (2014). Why does cryptographic software fail? A case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, New York, NY, USA. ACM.

Li, Y., Choi, D., et al. (2022). Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Liu, P., Liu, J., et al. (2024). Exploring ChatGPT's Capabilities on Vulnerability Management. In *33rd USENIX Security Symposium*, pages 811–828, Philadelphia, PA. USENIX Association.

Llama (n.d.). Llama. https://www.llama.com/. Retrieved September 21, 2024.

Masood, Z. and Martin, M. V. (2024). Beyond static tools: Evaluating large language models for cryptographic misuse detection.

Nadi, S., Krüger, S., et al. (2016). Jumping through hoops: why do Java developers struggle with cryptography APIs? In *Proceedings of the 38th International Conference on SE*, ICSE '16, page 935–946, New York, NY, USA. Association for Computing Machinery.

Ouh, E. L., Gan, B. K. S., et al. (2023). ChatGPT, Can You Generate Solutions for My Coding Exercises? An Evaluation on Its Effectiveness in an Undergraduate Java Programming Course. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, pages 54–60, New York, NY, USA. Association for Computing Machinery.

OWASP Benchmark (2016). OWASP Benchmark. https://owasp.org/www-project-benchmark/. Accessed May, 2024.

QwenLM (n.d.). QwenLM. https://qwenlm.ai/. Retrieved December 21, 2024.

Rahaman, S., Xiao, Y., et al. (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, page 2455–2472, New York, NY, USA. Association for Computing Machinery.

Redmiles, E. M., Warford, N., et al. (2020). A Comprehensive Quality Evaluation of Security and Privacy Advice on the Web. In *29th USENIX Security Symposium*, pages 89–108. USENIX Association.

Rostami, E. and Karlsson, F. (2024). Qualitative Content Analysis of Actionable Advice in Information Security Policies – Introducing the Keyword Loss of Specificity Metric. *Information & Computer Security*, 32(4):492–508.

Snyk (n.d.). Snyk Code — Code Security Analysis and Fixes - Developer First SAST. https://snyk.io/product/snyk-code/. Retrieved August 20, 2024.

Soot (2020). Soot. https://github.com/soot-oss/soot. Accessed: August 26, 2024.

Tabnine (n.d.). Tabnine. https://www.tabnine.com. Retrieved August 21, 2024.

Vallée-Rai, R., Co, P., et al. (1999). Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press.

Whitten, A. (2004). *Making Security Usable*. PhD thesis, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, USA.

Xia, Y., Xie, Z., et al. (2024). Exploring Automatic Cryptographic API Misuse Detection in the Era of LLMs.

Xie, J., Lipford, H. R., et al. (2011). Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 161–164, Los Alamitos, CA, USA. IEEE Computer Society.

Zhang, L., Chen, J., et al. (2019). CryptoREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices. In *22nd International Symposium on RAID*, pages 151–164, Chaoyang District, Beijing. USENIX Association.

Zhang, Y., Kabir, M. M. A., et al. (2023). Automatic Detection of Java Cryptographic API Misuses: Are We There Yet? *IEEE Transactions on Software Engineering*, 49(1):288–303.

APPENDIX

LLM Prompt. Large Language Models (LLMs) received standardized prompts for each test case from the three benchmarks to ensure their responses were consistent. The specific prompt used for each test case is shown in Listing 1. During the experiments, we kept the default settings for model hyperparameters like temperature, Top P, and frequency penalty to maintain each model's natural response style. Responses from GPT 4-o-mini were collected automatically through an API, while responses from Llama, Claude, and Gemini were gathered manually from its chat interface.

```
1 I want you to detect "
      Cryptographic misuses" in the
      given Java code by considering
      the cryptographic misuse
      definitions below.
2
3 Cryptographic misuses are
      deviations from best practices
      while incorporating
```

```
        cryptographic algorithms into
        your software that could
        potentially be exploited by an
        adversary. Broadly,
        cryptographic misuses can be
        referred to as bad programming
        practices that create
        vulnerabilities and are
        associated with design flaws
        and unsafe architectural
        choices.
4
5   Identify the following for each
        cryptographic misuse and
        include it in your response.
        Follow the same template for
        reporting multiple misuses.
6
7   1- Include a label as YES/NO if
        there are specifically
        cryptographic misuses in the
        given code or not.
8   2- Include the name of the method
        containing the misuse.
9   3- Specify a message highlighting
        the type of misuses. Provide a
        description in the explanation
        as to why it is highlighted as
        a misuse.
10  4- State the line from the code
        containing the misuses.
        Specifically mention the
        starting line number that is
        causing misuse.
11
12  Code:
13  // Code for Each test case is
        placed here
```

Listing 1: Prompt Used to Query LLMs

LLM Responses. The responses from the LLMs were fairly similar in format due to the structured prompt, though some variations were observed. For instance, in certain cases, the explanation for misuse from GPT 4-o-mini appeared under tags such as “Misuse Description”, while in others, it was labeled as “Explanation” or a similar term. An example of this can be seen in Listing 3, which shows the response for the CryptoAPI Bench test case listed in Listing 2. The misuse in this instance involves using the DES algorithm and the “DES” string to initialize the key. The detailed findings are discussed in Section 5.

```
1   public class BrokenCryptoABMCCase1
2   {
3       public static void main(String
            [] args) throws
            NoSuchPaddingException,
            NoSuchAlgorithmException,
            InvalidKeyException
4       {
5           BrokenCryptoABMC1 bc = new
                BrokenCryptoABMC1();
6           String crypto = "DES/ECB/
                PKCS5Padding";
7           String cryptokey = "DES";
8           bc.go(crypto, cryptokey);
9       }
10  }
```

Listing 2: CryptoAPI Bench Test Case: BrokenCryptoABMCCase1.java

```
1   Response:
2   1- **Label**: YES
3   2- **Method containing the misuse
        **: main
4   3- **Message highlighting the type
         of misuses**: Use of weak
       cryptographic algorithm and
       mode of operation
5      **Explanation**: The code uses
           DES (Data Encryption
           Standard) in ECB (Electronic
            Codebook) mode, which is
           considered weak and insecure
           . DES itself has been
           deprecated due to its small
           key size and susceptibility
           to brute-force attacks,
           while ECB mode exposes
           patterns in plaintext since
           identical plaintext blocks
           produce identical ciphertext
            blocks. This can lead to
           serious vulnerabilities,
           allowing attackers to
           perform cryptanalysis or
           deduce information about the
            plaintext. Best practices
           recommend using stronger
           algorithms (e.g., AES) and
           secure modes of operation (e
           .g., GCM or CBC with proper
           IV management).
6   4- **Line from the code containing
         the misuses**: 10-11 (Lines
       mentioning `String crypto = "
       DES/ECB/PKCS5Padding";` and `
       String cryptokey = "DES";`)
```

Listing 3: GPT 4-o-mini Response for BrokenCryptoABMCCase1.java