Cryptbara: Dependency-Guided Detection of Python Cryptographic API Misuses

Seogyeong Cho
Korea University
Republic of Korea
jsg8777@korea.ac.kr

Seungeun Yu
Korea University
Republic of Korea
spblue4422@korea.ac.kr

Seunghoon Woo*
Korea University
Republic of Korea
seunghoonwoo@korea.ac.kr

***Abstract*—We present Cryptbara, a precise approach for detecting Python cryptographic API misuses. Cryptographic APIs are widely used to ensure data security, but their improper use can inadvertently compromise the security of entire systems. Existing approaches often fail to capture how cryptographic objects are initialized and used across inter-procedural contexts, limiting their ability to detect context-dependent misuses. In contrast, the key innovation of Cryptbara lies in synergistically combining static dependency analysis with LLM reasoning guided by dependency context, enabling context-sensitive misuse detection. To this end, Cryptbara extracts intra- and inter-procedural dependencies from Python code and encodes them into context-rich prompts, allowing the LLM to perform semantically-aware analysis despite syntactic complexity. We evaluated Cryptbara on two benchmarks containing real-world cryptographic API misuses. Cryptbara achieved F1 scores of 95.43% and 84%, outperforming existing approaches that achieved at most 73.68% and 70.59% F1 scores, respectively. Cryptbara further demonstrated its practical impact by discovering previously unknown misuses in popular Python repositories, with 22 representative cases reported to and confirmed by maintainers.**

***Index Terms*—Python cryptographic API misuse, Dependency analysis, Misuse detection**

I. Introduction

Modern software systems heavily rely on cryptographic primitives to ensure data confidentiality, integrity, and authenticity [1]. Ironically, the cryptographic APIs can become sources of vulnerabilities when misused [2]. Developers often struggle to select secure configurations or understand the requirements of cryptographic APIs, thereby introducing vulnerabilities [3]–[5]. In particular, the Python ecosystem, widely adopted in server applications and AI systems, extensively utilizes cryptographic libraries and APIs [6]. However, analyzing correct cryptographic API usage in Python is particularly difficult, mainly due to the following two common aspects of cryptographic API usage in Python (see Section II-B).

- **Syntactic ambiguity.** Cryptographic parameters are often constructed across multiple functions or tied to dynamically resolved object types, making them difficult to analyze through static syntax or flow alone.
- **Semantic ambiguity.** Misuses such as insecure fallback logic or policy violations require a semantic understanding of code, which traditional analyzers cannot capture.

Previous efforts to detect cryptographic API misuses in Python have shortcomings, primarily due to their inability to fully capture the syntax and semantic complexity of Python code. For example, *LICMA* [7] relies on a simple rule-based approach. Although it performs limited backward data-flow analysis, it fails to capture deeper inter-procedural relationships. Cryptolation [8] defines more comprehensive rules and employs program slicing with inter-procedural data-flow analysis. However, it can detect only misuses explicitly covered by its specifications and does not fully account for Python's unique characteristics such as dynamically resolved objects. CryptoPyt [9] leverages taint analysis; however, it still lacks the semantic understanding necessary to identify context-dependent misuses, such as fallback logic (see Table I).

Our approach. We propose Cryptbara (CRYPTographic API misuse BARricAde), the first approach to integrate static dependency analysis with LLM-based reasoning for semantic-aware Python cryptographic API misuse detection.

The main novelty of Cryptbara lies in two key aspects: (1) a crypto-specific dependency analysis tailored to Python's dynamic context, and (2) a dependency-guided LLM semantic reasoning methodology that exploits these dependencies for accurate misuse detection. Instead of simply combining static analysis with an LLM-based approach, we enhanced both to better accommodate Python's unique characteristics.

Given an input Python codebase, Cryptbara begins by identifying cryptographic API calls. It then analyzes both intra- and inter-procedural dependencies by collecting related functions, including those that either call or are called by the identified APIs. Cryptbara focuses on two key syntactic ambiguities that prior approaches fail to handle effectively (see Section II-B): (1) *context fragmentation*, where critical information is distributed across multiple functions, and (2) *context-dependent code*, where the interpretation of API usage depends on the surrounding context. To this end, Cryptbara performs fine-grained dependency analysis across and within functions to extract critical dependencies (see Section III-A), such as parameter flows (to address context fragmentation) and receiver object resolution (to handle context-dependent code).

Using the extracted dependency information, Cryptbara constructs context-rich prompts for LLMs (see Section III-B). To further improve detection effectiveness, we consult official

[*] Corresponding author

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00133

TABLE I: Analysis capabilities of existing tools and Cryptbara.

Feature	Tool			
	LICMA	Cryptolation	CryptoPyt	Cryptbara
Intra-procedural analysis	✓	✓	✓	✓
Inter-procedural analysis	×	✓	✓	✓
Python-tailored analysis	×	△	✓	✓
Context-aware analysis	×	△	△	✓
Semantic-aware analysis	×	×	×	✓

documentation to extract security rules related to cryptographic API misuse (see Section III-B1). Cryptbara then composes prompts that incorporate the target code, relevant rules, and dependency information. This enables the LLM to reason about both the syntax and semantics of the code, facilitating more precise detection of cryptographic misuses.

Evaluation. We evaluated Cryptbara using two benchmark datasets: (1) *PyCryptoBench* [8] and (2) a *real-world Python cryptographic misuse dataset* that we constructed by analyzing security-relevant patch commits from open-source repositories (see Section IV-A1). Each benchmark contains diverse cryptographic misuses along with safe examples without any misuse.

When we compared Cryptbara with existing approaches (*i.e.*, *LICMA* [7] and Cryptolation [8]), Cryptbara outperformed both baselines across the two datasets. On *PyCryptoBench*, Cryptbara achieved 95.43% F1 score, while the existing approaches reached only 18.86% and 73.68%, respectively. On the *real-world dataset*, which contains significantly more complex code structures, Cryptbara achieved 84% F1 score, outperforming one approach that failed to detect any misuses and another that achieved 70.59% F1 score. As shown in Table I, the superior accuracy of Cryptbara results from its comprehensive analysis capabilities that existing tools lack, particularly Python-tailored features, context- and semantic-aware analysis (see Section IV-A).

To demonstrate the practicality of Cryptbara, we deployed it on widely used real-world Python repositories from GitHub. Consequently, Cryptbara detected 172 cryptographic API misuses. We reported 22 critical cases, which were confirmed as requiring urgent attention: four have been patched, and 11 cases are currently under discussion.

Contribution. We summarize our contributions below.

- We present Cryptbara, a novel approach for detecting Python cryptographic API misuses. The core technical contribution is a hybrid approach that combines fine-grained static dependency analysis with LLM-guided semantic reasoning to enable context-aware detection.
- On two benchmark datasets composed of diverse misuse cases, Cryptbara achieved 95.43% and 84% F1 scores, respectively, thereby outperforming existing approaches, which achieved up to 73.68% and 70.59% F1 scores.
- Cryptbara demonstrated practical impact by detecting 172 cryptographic misuses across popular Python repositories, with 22 critical cases reported to maintainers; four have been patched, and 11 remain under discussion.

Listing 1: Example of context fragmentation.

```
def get_iterations():
  return 10000
def derive_key():
  iters = get_iterations()
  return pbkdf2_hmac(‘sha256’, b‘pass’, b‘salt’, iters)
```

Listing 2: Example of context-dependence code.

```
def make_crypto():
  return AES.new(b”key1234567”, AES.MODE_CBC)
cipher = make_crypto()
cipher.update(b”secret”)
```

II. Problem Statement and Challenges

A. Problem statement

Cryptographic libraries are used to protect data, but the improper use of cryptographic APIs can introduce security threats. This is frequently encountered in real-world software systems where security is critical, putting sensitive user data at risk [10]. Therefore, we aim to detect Python cryptographic API misuses. This includes APIs invoked with improper parameters, the use of insecure or deprecated cryptographic functions, and the use of non-randomized or hardcoded values in security-critical operations (see Section III-B1).

B. Challenges

Detecting Python cryptographic API misuses is a non-trivial task due to the language's dynamic features and diverse library ecosystems. Even recent studies (*e.g.*, [8], [9]) have limitations that constrain their effectiveness in real-world scenarios.

Challenge I: Ambiguous syntax in cryptographic contexts. One major challenge in analyzing Python code lies in its syntactic flexibility. Security-critical values, such as cryptographic keys, initialization vectors (IVs), or iteration counts, are often not defined directly at API call sites. Instead, they are computed across helper functions or assembled through multi-step logic, resulting in *context fragmentation*. This makes it difficult for static analyzers to trace value origins and assess compliance with security policies. For example, Listing 1 shows a case where the iteration count used in `pbkdf2_hmac` is computed in a separate function. Because it enforces a policy requiring the iteration count to exceed a minimum threshold, static tools must perform inter-procedural analysis to verify that the requirement is met. However, existing approaches (*e.g.*, [8], [9]) often fail to capture such cases, as they are limited to tracking explicit flows and cannot reconstruct value derivation chains.

A second source of ambiguity arises from Python's dynamic typing and object-oriented design, which leads to *context-dependent resolution*. Cryptographic APIs often reuse method names across object types (*e.g.*, `update()` may refer to either a `cipher` or `hash` operation). Without resolving the object's type and construction path, static tools cannot determine whether a call represents encryption, hashing, or another operation. As shown in Listing 2, the method `update()` is invoked on an object returned by a function. Without knowing the return type of `make_crypto`, a static analyzer cannot associate this call with the correct misuse policy.

Listing 3: Example of semantically ambiguous code.

```python
def derive_key(input):
  return ... # generate key from input
def get_key(user_input):
  if user_input:
    return derive_key(user_input)
  return "default_key"
def encrypt(msg):
  user_input = input("Enter (leave blank to use default): ")
  key = get_key(user_input)
  cipher = AES.new(key.encode(), AES.MODE_ECB)
  return cipher.encrypt(msg)
```

Challenge II: Semantic ambiguity in cryptographic misuse. Even when the syntax ambiguity is addressed, determining whether a cryptographic API usage is secure often requires semantic understanding. Misuses such as hardcoded keys are not always syntactically obvious and often depend on control flow or implicit developer intent. These cases are difficult to capture with rule-based or taint-based analysis alone, as they are insufficient to understand the security context.

For example, in Listing 3, the function `get_key()` returns either a derived key from user input or a hardcoded key. Although the code looks structurally correct, it may perform encryption with a fixed key and `ECB` mode (both known to be insecure). Existing approaches may follow the data flow but miss the risk in the fallback logic, because they fail to fully interpret the control flow or the meaning of the values.

III. Design and Implementation of Cryptbara

In this section, we introduce Cryptbara, a hybrid approach for detecting Python cryptographic API misuses.

Resolving syntactic ambiguity is challenging, and the presence of semantic ambiguity further complicates our target problem. To this end, Cryptbara uses a synergistic approach that bridges the gap between static analysis and contextual understanding through dependency-enriched prompting.

Preliminary concepts. To facilitate understanding of our approach, we briefly explain several key terms used throughout this paper.

- **Data flow analysis** tracks how data values propagate through variables and expressions in a program, which is crucial for understanding the context and transformations of security-relevant data.
- **Control flow analysis** determines the possible execution paths of a program by analyzing the order in which statements and functions may be executed.
- **Intra-procedural analysis** examines control and data flows within a single function or procedure.
- **Inter-procedural analysis** extends the scope of analysis across multiple functions or procedures, capturing flows that span function boundaries.

These are essential for cryptographic misuse detection because they enable the identification of how cryptographic APIs are invoked, how their parameters are derived, and whether insecure data or configurations influence their usage.

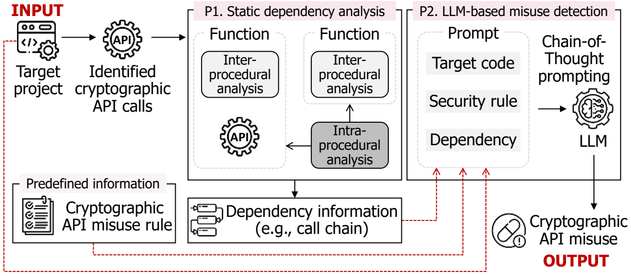

INPUT
Target project
Identified cryptographic API calls
P1. Static dependency analysis
Function
Inter-procedural analysis
Function
Inter-procedural analysis
Intra-procedural analysis
API
API
Predefined information
Cryptographic API misuse rule
Dependency information (e.g., call chain)
P2. LLM-based misuse detection
Prompt
Target code
Security rule
Dependency
Chain-of-Thought prompting
LLM
Cryptographic API misuse
OUTPUT

Fig. 1: High-level overview of Cryptbara.

Overview. Cryptbara comprises two phases: *static dependency analysis* (P1) and *LLM-based misuse detection* (P2). Figure 1 illustrates the high-level workflow of Cryptbara.

In P1, Cryptbara performs fine-grained static dependency analysis to extract intra- and inter-procedural relationships between cryptographic API calls and their arguments. By identifying dependencies related to cryptographic API usage, Cryptbara mitigates syntactic ambiguities.

In P2, Cryptbara feeds the extracted dependency context into LLMs via structured prompts. These prompts enable LLMs to reason about object semantics and configuration safety, thereby enabling more accurate detection of cryptographic misuses.

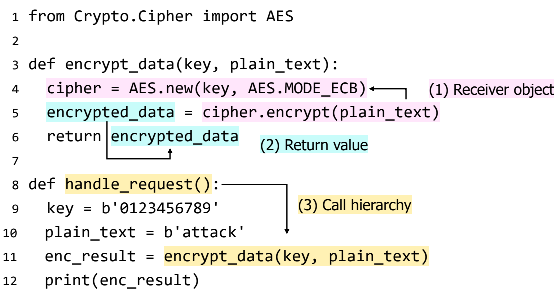
A. Static dependency extraction (P1)

In P1, Cryptbara examines the dependencies of the given Python code by using intra- and inter-procedural analysis. This dual approach enables comprehensive extraction of API usage context, data dependencies, and potential misuse patterns across both local and cross-functional scopes.

This phase begins by identifying Python cryptographic API calls in the codebase. Cryptbara filters method invocations corresponding to 204 commonly used cryptographic primitives, such as symmetric encryption (*e.g.*, `AES`) and key derivation (*e.g.*, `PBKDF2HMAC`). To compile this set, we manually collected frequently used classes and functions from popular Python libraries, including `pycryptodome`, `PyNaCl`, and `cryptography`. This can be performed straightforwardly using a function parser (*e.g.*, Joern parser [11]). The final list of 204 primitives was curated through an analysis of each library's official documentation, usage examples, and developer guides. Using primitive-level identification, Cryptbara ensures consistent detection in various Python cryptographic libraries.

1) Intra-procedural analysis: For the functions containing the identified cryptographic APIs, Cryptbara examines the following three key intra-procedural dependencies: *receiver object*, *return value*, and *call hierarchy*.

(1-1) Receiver object dependencies. Python APIs often follow object-oriented styles, where operations are performed by calling methods on receiver objects. For example, in the `encrypt_data` function in Figure 2, the `cipher` object is first created using `AES.new` (line #4), and the encryption is performed by invoking `encrypt` on `cipher` (line #5).



```
from Crypto.Cipher import AES

def encrypt_data(key, plain_text):
  cipher = AES.new(key, AES.MODE_ECB)
  encrypted_data = cipher.encrypt(plain_text)
  return encrypted_data

def handle_request():
  key = b'0123456789'
  plain_text = b'attack'
  enc_result = encrypt_data(key, plain_text)
  print(enc_result)
```

Fig. 2: Example of intra-procedural dependencies.

To understand how cryptographic objects are initialized, CRYPTBARA uses a *backward slicing* technique that traces the origin of receiver objects used in API calls.

(1) The backward slicing technique starts by identifying the cryptographic API call (*e.g.*, `cipher.encrypt`) and determining the receiver objects (*e.g.*, `cipher`).

(2) CRYPTBARA then follows the data flow in *reverse* to find where the variable was created. This includes locating assignment statements that call constructor functions (*e.g.*, `AES.new`) and checking the parameters passed during initialization (*e.g.*, `key` and `AES.MODE_ECB`).

CRYPTBARA leverages a code property graph (CPG), a unified representation that merges AST, control flow graphs, and data flow graphs, to identify data flows related to target functions. This enables precise tracking of both syntactic and semantic relationships in code, making it well-suited for our goal of extracting cryptographic API-related dependencies. Specifically, we utilized the Joern parser [11], which is widely adopted in related research, to extract CPGs from Python code.

For example, to identify the initialization of a receiver object such as `cipher` in `cipher.encrypt` (see Figure 2), CRYPTBARA first locates the API call node in the CPG and identifies the corresponding receiver variable. It then performs a backward traversal along data flow edges to find the assignment where the receiver was constructed (*e.g.*, `cipher = AES.new(...)`).

(1-2) Return value dependencies. Return values from cryptographic operations often contain sensitive data (*e.g.*, ciphertext or derived keys), and improper handling (*e.g.*, storage without protection) can undermine security guarantees. To detect insecure handling of cryptographic outputs, CRYPTBARA applies *forward data flow analysis* that tracks how return values from cryptographic API calls are used.

(1) The forward data flow analysis begins by identifying a statement where the output of a cryptographic API is assigned to a variable.

(2) CRYPTBARA then inspects the immediate statements that use this variable to determine how the value is used. Here, CRYPTBARA does not follow the full transitive data flow chain, but focuses on direct usages (*e.g.*, return statements, storage, or function arguments) appearing shortly after the assignment for simplicity and precision.

CRYPTBARA performs forward analysis over the CPG to trace data flows from cryptographic API invocations. This design prevents over-approximation due to long-range dependencies and highlights cases where cryptographic outputs are exposed too soon or without proper processing.

For example, the encrypted data (*i.e.*, `cipher.encrypt`) is assigned to `encrypted_data` in Figure 2. The variable is then immediately returned by the function without further processing. CRYPTBARA captures this single-step propagation using forward data flow analysis. CRYPTBARA can flag potential issues, such as insecure exposure of sensitive data, which may occur if encrypted values are returned or transmitted without additional protection (*e.g.*, integrity checks).

(1-3) Call hierarchy dependencies. CRYPTBARA examines how cryptographic operations are triggered by building intra-procedural call graphs. To do this, CRYPTBARA scans each function to extract all direct call expressions. For each call, the enclosing function is marked as the *caller*, and the called function name is recorded as the *callee*. The analysis does not cross function boundaries. As shown in Figure 2, the `handle_request` function calls `encrypt_data`, which contains the cryptographic API call. CRYPTBARA identifies this relationship by analyzing the body of `handle_request` and extracting the call to `encrypt_data`. This enables detection of cryptographic usage that occurs indirectly through one or more layers of internal function calls.

2) Inter-procedural analysis: Although intra-procedural analysis captures critical information within function boundaries, it has limitations in tracking data flows across functions and fully resolving context fragmentation. Therefore, CRYPTBARA performs inter-procedural analysis to extract three key dependencies for each identified cryptographic call: *parameter propagation*, *constant usage*, and *call chain*.

(2-1) Parameter propagation dependency. CRYPTBARA first examines how security-sensitive data propagates through multiple functions, including parameter passing, global variable access, and object attribute manipulation. Specifically, by traversing the AST of each function related to cryptography API calls, CRYPTBARA identifies references where cryptographic values move into or out of function scopes.

(1) For each parameter at function call sites, it performs backward tracing within the AST to determine the origin of values (*i.e.*, whether from local assignments, global variables, or object fields).

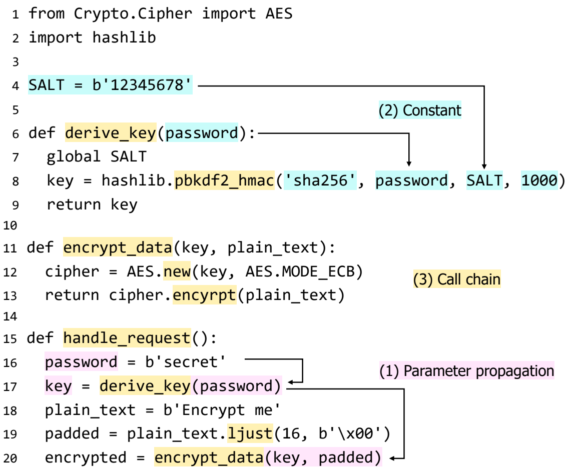
(2) CRYPTBARA then leverages caller-callee relationships (*i.e.*, identified by call hierarchy inspection within intra-procedural analysis) to follow these parameter flows across multiple function call layers, reconstructing both direct and indirect propagation paths.

The extracted parameter propagation dependencies ensure full visibility of how sensitive cryptographic values are passed, reassigned, or reused throughout the codebase.

For example, in Figure 3, CRYPTBARA identifies that `key` (*i.e.*, a parameter passed to `encrypt_data` in line #20) is

TABLE II: Summary of grouped cryptographic API misuse rules.

Group	ID	Category	Rule name	Checkpoints	Severity
Algorithm selection	R1	Symmetric encryption	Use secure and modern symmetric ciphers	Secure algorithm, sufficient key size	High
	R2	Asymmetric encryption	Use strong asymmetric key sizes	RSA $\geq$ 2048 bits, ECC $\geq$ 256 bits	High
	R3	Hash function	Avoid weak hash functions	Use SHA-256 or higher	High
	R4	Mode of operation	Avoid insecure block cipher modes	Use authenticated or randomized modes	High
Key and randomness management	R5	Key management	Avoid hardcoded or static keys	Keys should not be constant or predictable	High
	R6	PRNG quality	Use cryptographically secure PRNGs	Avoid `random` for secure keys	High
	R7	Seed management	Avoid predictable PRNG seeds	Use entropy-based seeding	Medium
Parameter and component management	R8	IV management	Avoid static IVs	IV should be randomized for each use	High
	R9	Salt management	Avoid static salts in PBE	Salt should be unpredictable	High
	R10	PBE iteration count	Use sufficient iteration count for PBE	#Iteration $\geq$ 100,000	High
Protocol and configuration security	R11	Secure configuration mode	Use authenticated cipher modes	Include MAC or AEAD mode	Medium



```
from Crypto.Cipher import AES
import hashlib

SALT = b'12345678'

def derive_key(password):
  global SALT
  key = hashlib.pbkdf2_hmac('sha256', password, SALT, 1000)
  return key

def encrypt_data(key, plain_text):
  cipher = AES.new(key, AES.MODE_ECB)
  return cipher.encyrpt(plain_text)

def handle_request():
  password = b'secret'
  key = derive_key(password)
  plain_text = b'Encrypt me'
  padded = plain_text.ljust(16, b'\x00')
  encrypted = encrypt_data(key, padded)
```

Fig. 3: Example of inter-procedural dependencies.

the return value of the `derive_key` function. It further traces that this value is obtained by calling `derive_key` with the `password` parameter, which is a local variable defined in `handle_request`. CRYPTBARA closely analyzes how parameters are passed along in this manner. In addition, it detects that the `SALT` used in `derive_key` originates from a global variable.

(2-2) Constant dependency. Cryptographic API misuses frequently originate from the use of hardcoded constants. Therefore, CRYPTBARA scans AST nodes to detect literal values commonly associated with cryptographic misuse (*e.g.*, fixed encryption keys). CRYPTBARA particularly focuses on how literal values are employed as arguments in cryptographic API calls and evaluates their effect on cryptographic operations. For example, in Figure 3, CRYPTBARA identifies that `password`, `SALT`, and `plain_text` are all hardcoded values. In addition, CRYPTBARA identifies that both constant values in the `derive_key` function (*i.e.*, `SALT` and `password`) are used in the call to `hashlib.pbkdf2_hmac`. This information is later used to assist in detecting potential API misuses.

(2-3) Call chain dependency. Finally, CRYPTBARA reconstructs complete call chains by expanding caller-callee relationships beyond the intra-procedural level, revealing how cryptographic operations are invoked through inter-procedural function calls. This analysis begins by building a function call graph from previously extracted caller-callee pairs. Starting from identified entry points or top-level functions, CRYPTBARA recursively traverses the graph to enumerate all possible execution paths leading to cryptographic API calls. The resulting call chains expose the execution sequences directing cryptographic operations, allowing CRYPTBARA to verify whether these functions are invoked along intended control flows or through unexpected paths.

Notably, CRYPTBARA does not strictly separate intra- and inter-procedural analysis, as they often interact in practice. For example, a constant may appear within a single function (intra-procedural), while a receiver object might be passed in as a parameter (inter-procedural). Rather than explicitly distinguishing the two techniques, CRYPTBARA leverages both types of information in combination to analyze the context surrounding cryptographic API calls.

B. LLM-based misuse detection (P2)

CRYPTBARA then employs an LLM to detect cryptographic API misuses by leveraging the structured context extracted during static dependency analysis.

1) Rules for cryptographic API misuse: Cryptographic misuse involves subtle implementation details and contextual factors that can lead to serious security vulnerabilities. To address this issue, we have developed a *structured rule* framework grounded in established security principles from OWASP, NIST, and PKCS [12]–[14]. Each rule specifies the target cryptographic primitive or API, the misuse condition (*e.g.*, insecure mode, weak key length, static salt), and the checkpoints needed to determine misuse, along with the rationale and references to the relevant standards. The framework covers major categories of cryptographic operations, ensuring both broad coverage and context-aware detection.

We consider the following four functional categories.

(1) **Algorithm selection.** Using insecure algorithms (*e.g.*, weak hash algorithms) can critically undermine the entire system. This category ensures the use of modern and recommended primitives.

TABLE III: Misuse and safe examples of cryptographic APIs.

ID	Category	Case	Example
R1	Symmetric encryption	Misuse	`DES.new(key)`
		Safe	`AES.new(key, AES.MODE_GCM)`
R2	Asymmetric encryption	Misuse	`RSA.generate_private_key(1024)`
		Safe	`RSA.generate_private_key(2048)`
R3	Hash function	Misuse	`hashlib.md5(), SHA1.new()`
		Safe	`hashlib.sha256(), SHA3_256.new()`
R4	Mode of operation	Misuse	`AES.new(key, AES.MODE_ECB)`
		Safe	`AES.new(key, AES.MODE_GCM)`
R5	Key management	Misuse	`key = b”my_hardcoded_key”`
		Safe	`key = os.urandom(32)`
R6	PRNG quality	Misuse	`random.random()`
		Safe	`os.urandom()`
R7	Seed management	Misuse	`random.seed(time.time())`
		Safe	`random.seed(os.urandom(16))`
R8	IV management	Misuse	`iv = b”0000000000000000”`
		Safe	`iv = os.urandom(16)`
R9	Salt management	Misuse	`salt = b‘salt1234’`
		Safe	`salt = os.urandom(16)`
R10	PBE iteration count	Misuse	`iterations=1000`
		Safe	`iterations=100000`
R11	Secure configuration	Misuse	`ChaCha20.new()` (no MAC)
		Safe	`ChaCha20_Poly1305.new()`
R12	TLS configuration	Misuse	`requests.get(url, verify=False)`
		Safe	`requests.get(url)`

(2) **Key and randomness management.** Secure key generation and strong randomness are fundamental to preventing brute-force and prediction-based attacks. According to NIST SP 800-133 [14], all cryptographic keys should be derived from an approved random bit generator, ensuring sufficient entropy and unpredictability.

(3) **Parameter and component management.** Misconfigured or hardcoded parameters (*e.g.*, IVs and iteration counts) often lead to subtle but severe vulnerabilities, necessitating strict validation. For example, according to the recent OWASP guidelines [12], [13], `PBKDF2` should use as many iterations as practical—specifically, 600,000 for `PBKDF2-HMAC-SHA256` and 210,000 for `PBKDF2-HMAC-SHA512`.

(4) **Protocol and configuration security.** Even when strong primitives are used, insecure protocols or improper configurations can break end-to-end security guarantees.

Based on the previously referenced documents and the four categories, we selected 11 rules related to cryptographic API misuse. Table II summarizes the selected 11 misuse rules, and Table III presents both misuse and safe examples. For each rule, we specify the essential checkpoints that must be verified. These checkpoints will later serve as elements within prompts when detecting misuses using LLM-based methods.

Compared to recent taint analysis-based studies on detecting cryptographic API misuse in Python [9], rules that are not directly related to cryptographic APIs (*e.g.*, use of JSON Web Tokens) have been excluded, and some rules have been consolidated (*e.g.*, secure configuration mode). Although the overall rule set is not significantly different—aside from a stronger emphasis on cryptographic API misuse, CRYPTBARA employs a distinct approach in the misuse detection phase (*e.g.*, combining dependency analysis with LLM-based reasoning), setting it apart from existing techniques.

```
*Prompt*: You are a security analyst. You are provided with a rule, a few
examples of violations, target Python code, and dependency information.
Determine whether there exists any cryptographic API misuse.
Important constraints:
 - Do not assume that the code must contain a misuse. It may or may not.
 - Only consider misuses within the given rule.
Misuse rules: {rules}                   // Enumerate 11 rules
Code: {target_code}                     // Target Python code
Dependency: {dependencies}              // Enumerate every dependency
*Output*: {misuses}                     // in a JSON format
```

Fig. 4: Example structure of a prompt passed to the LLM.

2) Prompt construction: CRYPTBARA constructs each LLM prompt by integrating static analysis outputs into a structured format.

A prompt contains the following three elements.

(1) ***Target code snippet.*** The specific code region containing cryptographic API calls.

(2) ***Security rule.*** Definition of misuse patterns to evaluate.

(3) ***Dependency.*** Intra- and inter-procedural dependencies.

Figure 4 shows an example prompt structure, which incorporates all identified dependencies along with the selected rules. Our prompt structure restricts the evaluation scope to specific vulnerability patterns. For example, when analyzing nonce reuse, the prompt directs LLM's focus solely on nonce handling rather than examining peripheral security concerns.

In the final step, if the LLM classifies a code instance as a cryptographic API misuse, CRYPTBARA regards it as a misuse case. If the model does not raise any concerns, CRYPTBARA treats the usage as benign and proceeds without intervention. Although various LLM models are available, in our experiments, CRYPTBARA employs `GPT-4o-mini`, a widely used model known for delivering strong performance without fine-tuning [15], [16].

C. *Implementation of* CRYPTBARA

CRYPTBARA consists of approximately 1,900 lines of Python code, excluding external libraries. CRYPTBARA comprises two modules: *static analyzer* (for P1) and *misuse detector* (for P2). The static analyzer examines dependencies in the input codebase. It utilizes Joern's Python preprocessing pipeline (`pysrc2cpg`) to generate a CPG [11]. Inter-procedural dependency analysis is performed using Python's built-in `AST` module. The misuse detector leverages LLM to detect cryptographic API misuses. It combines predefined rules with relevant code snippets, and is implemented using the `OpenAI` `GPT-4o-mini`. Each prompt was issued five times, and the decision was determined by majority vote. The evaluation of prompt components and LLM backends is presented in Section IV-B.

IV. Evaluation

In this section, we evaluate Cryptbara based on the following four research questions.

- **RQ1: Accuracy.** How accurately does Cryptbara detect cryptographic API misuses compared to state-of-the-art static analysis tools? (Section IV-A)
- **RQ2: Effectiveness.** How do prompt components and the choice of LLM backend affect the effectiveness of Cryptbara? (ablation study; Section IV-B)
- **RQ3: Performance.** How efficient is Cryptbara in detecting cryptographic API misuses? (Section IV-C)
- **RQ4: Practicality.** Can Cryptbara find unknown cryptographic API misuses in the wild? (Section IV-D)

Experiments were conducted on a macOS machine equipped with an Apple M2 chipset (8-core CPU, 16GB RAM).

A. Accuracy

1) Benchmark dataset: To evaluate the accuracy, we conduct experiments using two benchmark datasets: (1) the publicly available *PyCryptoBench* [8] and (2) our own curated dataset of *real-world Python cryptographic API misuses*.

We selected *PyCryptoBench* to enable fair and reproducible comparisons, as it is widely adopted in recent studies and provides labeled misuse and safe examples across diverse misuse categories. Because Cryptbara is designed to verify the secure use of cryptographic APIs, we excluded test cases unrelated to such APIs, including those involving serialization or regular expressions. As a result, we evaluated 636 files (out of 1,836): 96 labeled as vulnerable and 540 as safe. However, *PyCryptoBench* consists mostly of short code snippets (*e.g.*, individual functions) and lacks real-world complexity.

Therefore, we constructed an additional dataset from popular Python repositories to evaluate Cryptbara on real-world code, focusing on complex, multi-function cryptographic misuses. We constructed the real-world dataset by combining the *known* and *hidden* misuse fix commits.

- **Known misuse set.** We first scanned all Python-related CVEs to identify cases involving cryptographic misuse CWEs (*e.g.*, CWE-330: Use of Insufficiently Random Values) with available patch commits on GitHub, and then collected the patch commits [17]–[19].
- **Hidden misuse set.** We manually searched GitHub using cryptography-related keywords (*e.g.*, "`hardcoded key`"), applying a language filter for Python, and collected commits that fixed insecure cryptographic API usage [20].

Each commit was labeled based on its pre- (insecure) and post-patch (safe) state. However, in some cases, post-patch code remained insecure and was fixed in a subsequent commit. We therefore manually rechecked all safe-labeled code to finalize the insecure and safe labels. Consequently, we collected **26 source files with misuses and 21 secure files** from more than 20 real-world repositories, including `mindsdb` and `ajenti` repositories. Among the 26 misuse files, 5 cases are from the known misuse set, and the remaining 21 cases are from the hidden misuse set. These files encompass four categories of cryptographic misuses: algorithm (8), key/randomness (15), parameter/component (2), and protocol (1). In particular, the collected files contain 236 distinct Python functions, which is not significantly smaller than those in the first benchmark.

2) Methodology: We compared Cryptbara with two state-of-the-art approaches with publicly available and reproducible implementations: *LICMA* [7] and Cryptolation [8]. Both tools are static analysis-based and aim to identify Python cryptographic API misuses. In particular, Cryptolation can identify various categories of cryptographic API misuses, making it a suitable subject for comparison. Other tools, such as CryptoPyt [9], were unavailable despite our request. We provide an indirect comparison with CryptoPyt in Section IV-A5.

We used four standard metrics: correctly identifying a misuse as a *true positive* (`TP`), incorrectly labeling a misuse as safe as a *false negative* (`FN`), incorrectly flagging a safe case as a misuse as a *false positive* (`FP`), and correctly identifying a safe case as a *true negative* (`TN`). To identify FPs and FNs, for *PyCryptoBench*, we used the provided misuse and safe labels as-is. For the *real-world dataset*, we used the misuse and safe labels that we had previously established for each commit.

We then computed *precision* (`P = #TP/(#TP+#FP)`), *recall* (`R = #TP/(#TP+#FN)`), and *F1 score* (`(2*P*R)/(P+R)`) to assess the overall effectiveness of each tool.

3) Comparison on PyCryptoBench.: Table IV summarizes the measurement results. Cryptbara achieved 95.43% F1 score, whereas the existing approaches achieved 18.87% and 73.68% F1 scores, respectively. *LICMA*, which relies on simple selected rules, failed to identify many misuse cases. Cryptolation showed particular weakness in detecting cryptographic misuses that require understanding Python's dynamic context. In the G1-3 group in Table IV, where context-dependent analysis is critical, Cryptolation achieved only 41.67% recall. Both approaches struggled to detect syntactically or semantically ambiguous misuses, yielding many FNs.

Cryptbara successfully identified all cryptographic misuses, except for two cases in the G1–3 group. These exceptions involved parameter management issues that were not detected because the misuse pattern was not included in the selected rule set. Notably, Cryptbara achieved the highest F1 score across all groups. It identified all misuse cases detected by *LICMA*. However, Cryptbara was not a superset of Cryptolation. This discrepancy arose because Cryptolation considered certain misuse rules that Cryptbara did not incorporate. Seven FPs arose from safe code that merely imported potentially dangerous libraries (*e.g.*, `import md5`, used in Python 2). Despite the absence of any misused API, the LLM flagged them as potentially risky.

4) Comparison on real-world dataset.: Table V presents a summary of the accuracy results of the three tools. Similar to the first benchmark, Cryptbara achieved the highest F1 score of 84%, outperforming existing techniques, which achieved 0% and 70.59% F1 scores, respectively.

TABLE IV: Accuracy evaluation results on *PyCryptoBench*.

IDX	Group*	Tool	#TP	#FP	#FN	#TN	Precision	Recall	F1 score
G1-1	*Algorithm selection*	*LICMA*	10	**0**	38	**468**	**100.00%**	20.83%	34.48%
		Cryptolation	31	**0**	17	**468**	**100.00%**	64.58%	78.48%
		Cryptbara	**48**	6	**0**	462	88.89%	**100.00%**	**94.12%**
G1-2	*Key and randomness management*	*LICMA*	0	**0**	24	**12**	0.00%	0.00%	0.00%
		Cryptolation	15	**0**	9	**12**	**100.00%**	62.50%	76.92%
		Cryptbara	**24**	1	**0**	11	96.00%	**100.00%**	**97.96%**
G1-3	*Parameter and component management*	*LICMA*	0	**0**	24	**60**	0.00%	0.00%	0.00%
		Cryptolation	10	**0**	14	**60**	**100.00%**	41.67%	58.82%
		Cryptbara	**22**	**0**	**2**	**60**	**100.00%**	**91.67%**	**95.65%**
Total results		*LICMA*	10	**0**	86	**540**	**100.00%**	10.42%	18.87%
		Cryptolation	56	**0**	40	**540**	**100.00%**	58.33%	73.68%
		Cryptbara	**94**	7	**2**	533	93.07%	**97.92%**	**95.43%**

**Protocol and configuration security* was excluded because it is not present in this benchmark dataset.

TABLE V: Accuracy evaluation results on *real-world dataset*.

IDX	Group	Tool	#TP	#FP	#FN	#TN	Precision	Recall	F1 score
G2-1	*Algorithm selection*	*LICMA*	0	**0**	8	**5**	0.00%	0.00%	0.00%
		Cryptolation	3	1	5	4	75.00%	37.50%	50.00%
		Cryptbara	**7**	2	**1**	3	**77.89%**	**87.50%**	**82.35%**
G2-2	*Key and randomness management*	*LICMA*	0	**0**	15	**11**	0.00%	0.00%	0.00%
		Cryptolation	**13**	5	**2**	6	72.22%	**86.67%**	78.79%
		Cryptbara	11	1	4	10	**91.67%**	73.33%	**81.48%**
G2-3	*Parameter and component management*	*LICMA*	0	**0**	2	**3**	0.00%	0.00%	0.00%
		Cryptolation	1	**0**	1	**3**	100.00%	50.00%	66.67%
		Cryptbara	**2**	**0**	**0**	**3**	**100.00%**	**100.00%**	**100.00%**
G2-4	*Protocol and configuration security*	*LICMA*	0	**0**	1	**2**	0.00%	0.00%	0.00%
		Cryptolation	**1**	1	0	1	50.00%	**100.00%**	66.67%
		Cryptbara	**1**	**0**	**0**	**2**	**100.00%**	**100.00%**	**100.00%**
Total results		*LICMA*	0	0	26	21	0.00%	0.00%	0.00%
		Cryptolation	18	7	8	14	72.00%	69.23%	70.59%
		Cryptbara	**21**	**3**	**5**	**18**	**87.50%**	**80.77%**	**84.00%**

Due to the complexity of real-world Python code syntax, *LICMA*'s simple rule-based approach failed to detect any misuse cases. Cryptolation performed better than *LICMA* but missed cases involving context fragmentation. It also produced seven FPs due to misinterpreting complex code.

Cryptbara achieved an F1 score of 84% on the real-world dataset, the highest among all tools. Although it effectively extracted dependency information even from complex cases, some FPs were caused by misinterpretation during the LLM-based detection phase. The five FNs were caused by misuse patterns that were not covered by our predefined rule set. In particular, most inter-procedural cases were concentrated in G2-1, whereas G2-2 mainly included simpler, rule-based issues such as seed management. Therefore, although Cryptbara achieved the best F1 score in both subsets, Cryptolation identified more TPs in G2-2.

5) Indirect comparison with CryptoPyt: CryptoPyt [9] is a state-of-the-art misuse detection approach that relies on rule-based static analysis. However, we noted that CryptoPyt (1) is limited to detecting misuses within predefined taint specifications, (2) defines rules that cover broad security concerns (*e.g.*, network or authentication configuration) rather than focusing specifically on cryptographic API misuses, and (3) does not fully account for Python's dynamic characteristics such as context-dependent object resolution.

In contrast, Cryptbara performs dependency analysis tailored to Python's features, applies crypto-specific rules, and leverages dependency-guided LLM semantic reasoning, which we believe contributes to higher effectiveness in identifying cryptographic API misuses.

Answer to RQ1. Cryptbara successfully identified Python cryptographic API misuses with higher accuracy than existing approaches in both benchmarks. Its comprehensive dependency analysis and semantic-aware LLM-based approach overcame the limitations of prior methods.

TABLE VI: Accuracy evaluation across different prompt design.

Tool	#TP	#FP	#FN	#TN	Precision	Recall	F1 score
Zero-shot	70	**1**	26	**539**	**98.59%**	72.92%	83.83%
Few-shot (FS1)	65	2	31	538	97.01%	67.71%	79.75%
Few-shot (FS2)	89	7	7	533	92.71%	92.71%	92.71%
Chain-of-Though	**94**	7	**2**	533	93.10%	**97.92%**	**95.43%**

B. Effectiveness

We then evaluate the effectiveness of CRYPTBARA using the *PyCryptoBench* benchmark dataset.

1) Prompt design: We first evaluate the effectiveness of CRYPTBARA's prompt design by comparing four strategies.

1) **Zero-shot prompting.** The LLM is presented with only the code, without any accompanying rule or context. This evaluates the model's inherent ability to identify misuses based solely on its pre-trained knowledge.
2) **Few-shot prompting with dependencies (FS1).** The LLM is provided with the code and its dependencies, but without misuse rules. This evaluates the contribution of dependency information alone.
3) **Few-shot prompting with rules (FS2).** The LLM receives the code and misuse rules, but without dependencies. This setup evaluates the effectiveness of structured rules without contextual dependency information.
4) **Chain-of-Thought (CoT) prompting.** The LLM is provided with both misuse rules and dependencies, guiding it through step-by-step reasoning. This evaluates the full potential of CRYPTBARA's hybrid approach by combining all available contextual information.

Result analysis. Table VI summarizes the accuracy of cryptographic API misuse detection across different prompt designs, and Figure 5 illustrates the accuracy differences of prompt designs compared to zero-shot prompting.

Although zero-shot prompting resulted in the fewest FPs, it also failed to detect many misuses, because the model was not given explicit misuse rules and dependencies, making it difficult for the LLM to identify violations. Interestingly, FS1 resulted in a lower F1 score than the zero-shot baseline. We hypothesize that, in the absence of explicit rules, the LLM struggles to interpret the rich dependency information, whereas in the zero-shot setting, the model may lean on its pre-trained knowledge to make more coherent judgments. In contrast, FS2 significantly outperformed both zero-shot and FS1. Although the presence of rules helped clarify what to look for, the lack of structural context yielded seven FNs. Finally, the CoT setting, which combines both rules and dependency information, achieved the highest F1 score, demonstrating the effectiveness of integrating both semantic criteria and structural context.

2) LLM backend: Next, we evaluate the accuracy of CRYPTBARA using different LLM backends to analyze how the choice of backend affects its effectiveness. We compared three different LLM backends under the same prompting design (CoT): `OpenAI`'s (1) `GPT-4o-mini`, (2) `GPT-3.5-turbo`, and `Meta`'s latest open source model, (3) `LLaMA3` (`LLaMA3-70B-8192`). We selected these models because they are widely available and achieve strong performance without fine-tuning [15], [16], [21]. This reflects our goal of enabling practical adoption in real-world development environments.

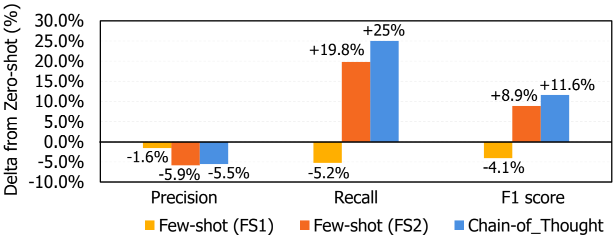


Fig. 5: Accuracy difference compared to zero-shot prompting.

TABLE VII: Accuracy on different LLM backends.

Group*	Backend	#TP	#FP	#FN	#TN	Precision	Recall	F1 score
G1-1	`GPT-3.5-turbo`	**48**	**6**	**0**	**462**	**88.89%**	**100.00%**	**94.12%**
	`LLaMA3`	5	10	43	458	33.33%	10.42%	15.87%
	`GPT-4o-mini`	**48**	**6**	**0**	**462**	**88.89%**	**100.00%**	**94.12%**
G1-2	`GPT-3.5-turbo`	7	**0**	17	**12**	**100.00%**	29.17%	45.16%
	`LLaMA3`	17	**0**	7	**12**	**100.00%**	70.83%	82.93%
	`GPT-4o-mini`	**24**	1	**0**	11	96.00%	**100.00%**	**97.96%**
G1-3	`GPT-3.5-turbo`	0	**0**	24	**60**	0.00%	0.00%	0.00%
	`LLaMA3`	7	1	17	59	87.50%	29.17%	43.75%
	`GPT-4o-mini`	**22**	**0**	**2**	**60**	**100.00%**	**91.67%**	**95.65%**
Total	`GPT-3.5-turbo`	55	**6**	41	**534**	90.16%	57.29%	70.06%
	`LLaMA3`	29	11	67	529	72.50%	30.21%	42.65%
	`GPT-4o-mini`	**94**	7	**2**	533	**93.07%**	**97.92%**	**95.43%**

*We use the same group indices (IDX) as defined in Table IV.

Result analysis. Table VII presents the experimental results. We observed that `GPT-4o-mini` achieved the highest F1 score of 95.43%, while `GPT-3.5-turbo` and `LLaMA3` reached 70.06% and 42.65% F1 scores, respectively. Notably, `GPT-4o-mini` achieved the highest F1 score across all three groups (see Table IV). `GPT-3.5-turbo` showed comparable performance to `GPT-4o-mini` in Group G1-1, but its accuracy dropped significantly in the other groups. This may be because `GPT-3.5-turbo` is more prone to overlooking subtle misuse patterns or less capable of reasoning over complex prompt structures, even when provided with sufficient dependency information. In contrast, `LLaMA3` performed reasonably well in G1-2 and G1-3, but detected only 5 out of 48 misuse cases in G1-1. This may be due to its limited ability to integrate the provided dependency context into accurate misuse reasoning, particularly in cases requiring precise interpretation of control and data flow.

Answer to RQ2. Prompt design evaluation showed that adding misuse rules and dependency information to the prompt significantly improved accuracy. This highlights the effectiveness of CRYPTBARA's CoT-based prompting strategy. In the LLM backend comparison, `GPT-4o-mini` outperformed others, achieving the highest F1 score.

C. *Performance*

We then measured the processing time per file to evaluate CRYPTBARA's performance. Each file in the two benchmarks contains an average of 54 lines of Python code. For each file, CRYPTBARA took an average of 41 s to extract dependencies using the static analyzer, and 3 s to obtain a response from the misuse detector with a single prompt (15 s for 5 prompts). Therefore, on average, CRYPTBARA required 56 s to identify cryptographic API misuses in a single Python file. Although the processing time varied slightly depending on the number of lines of code, the difference was not significant (*i.e.*, within 10 s). Despite the relatively small number of lines in the file, these results show that CRYPTBARA achieves precise analysis with efficient processing.

> **Answer to RQ3.** CRYPTBARA can detect cryptographic API misuses in a single Python file in under one minute on average, demonstrating its practical effectiveness.

D. *Cryptographic API misuses in the wild*

Finally, we applied CRYPTBARA to real-world, popular Python repositories to identify previously unknown cryptographic API misuses. We collected codebases from GitHub's popular Python repositories with more than 5,000 stargazers and applied CRYPTBARA to analyze them.

As a result, CRYPTBARA identified **172 potential cryptographic API misuses** across 34 repositories. The detected misuses varied in type, including hardcoded keys and IVs, inappropriate iteration counts, and the use of insecure `AES` modes (*e.g.*, `CBC`). After manual analysis, we reported 22 of these cases as potentially exploitable vulnerabilities. As of Aug 2025, four of the reported issues have been confirmed and patched, and 11 are under discussion. In addition, seven cases were confirmed as low-risk and not planned for patching.

Listing 4 illustrates a real-world misuse that was patched following our report. This was found in `DB-GPT` (16.5 K GitHub stars as of May 2025), an AI-native data application framework. The misuse arose from configuring `PBKDF2-HMAC-SHA256` with a low iteration count (100,000), which falls short of the recommended value: according to security guidelines [13], a secure configuration should use at least 600,000 iterations. CRYPTBARA detected this issue, and upon reporting it to the developers, the misuse was promptly patched.

Listing 5 shows another real-world case. This misuse involves `AES-CBC` encryption with an IV deterministically derived from user input (*i.e.*, hardcoded). Because the same IV is reused, identical plaintexts yield identical ciphertexts, violating semantic security. Detecting this requires interprocedural analysis: the IV is set in `init` but used in `encrypt()`. CRYPTBARA identifies the static IV via dependency tracking, and the LLM correctly reasons that reusing it in `CBC` mode is insecure, even though the IV is not a constant. We reported this issue to the development team, and they acknowledged the vulnerability. As the patch has not yet been applied, we redact specific details to prevent potential exploitation.

Listing 4: Real-world misuses patched by our report.

```
def _generate_key_from_password(
  password: bytes, salt: Optional[Union[tr, bytes]] = None
):
  if salt is None:
    salt = os.urandom(16)
  elif isinstance(salt, str):
    salt = salt.encode()
  kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(), length=32, salt=salt,
-   iterations=100000,
+   iterations=800000,
  )
  key = base64.urlsafe_b64encode(kdf.derive(password))
  return key, salt
```

Listing 5: Another real-world misuse case. As the patch has not yet been released despite our disclosure request, we redact package and class identifiers to prevent potential exploitation.

```
def pad(msg) :
  pad_len = 16 - len(msg) % 16
  return msg + bytes([pad_len] * pad_len

class LabelEncryptor :
  def __init__(self, user_id: str):
    key = hashlib.sha1(user_id.encode()).digest()[:16]
    self.iv = hashlib.sha256(key).digest()[:16]
    self.key = key
  def encrypt(self, msg: str) -> bytes:
    cipher = AES.new(self.key, AES.MODE_CBC, self.iv)
    return cipher.encrypt(pad(msg.encode())
```

> **Answer to RQ4.** CRYPTBARA successfully identified previously unknown real-world cryptographic misuses, including cases that were acknowledged and fixed by developers, demonstrating its practical effectiveness.

V. DISCUSSION

1) LLMs without fine-tuning: In designing CRYPTBARA, we intentionally avoided both fine-tuning LLMs and incorporating retrieval-augmented generation (RAG) mechanisms. This design choice was grounded in a desire for practicality and generalizability. Fine-tuning, while potentially improving performance, introduces risks of overfitting to narrow patterns. Similarly, RAG frameworks require additional components such as vector databases, making them more complex to deploy and maintain. In contrast, CRYPTBARA relies on structured context-rich prompts. Our evaluation shows that this lightweight design achieves high accuracy in various misuse patterns, supporting the effectiveness of prompt-based reasoning when combined with structured code dependencies.

2) Limitations and future work.: First, CRYPTBARA struggles to analyze indirect calls. When cryptographic objects are created through indirect calls, CRYPTBARA may fail to reconstruct the full call chain. To address this limitation, we plan to enhance CRYPTBARA's interprocedural analysis to better resolve indirect object constructions. Second, although CRYPTBARA effectively detects a wide range of real-world cryptographic misuses, there remain cases that are inherently difficult to analyze statically. For example, API usage hidden behind dynamic constructs such as `getattr` or reflective calls often prevents precise identification of misuse. Addressing such dynamic behaviors, possibly through hybrid or runtime-assisted analysis, remains a key direction for future work. Last, CRYPTBARA assumes that source code is available and analyzable. However, in real-world scenarios, obfuscated logic or the use of native extensions (*e.g.*, C/C++ modules) may limit static analysis and reduce detection accuracy.

3) Threats to validity: Although we evaluate CRYPTBARA using both the widely adopted *PyCryptoBench* and an additional benchmark curated from real-world misuse cases, these may not fully represent the entire spectrum of cryptographic API misuses in Python. Second, the manual labeling process is used to establish ground truth for evaluation. Although we followed established guidelines and performed cross-validation to reduce bias, human error may still affect the accuracy of the labels. Third, detection was evaluated using manually labeled ground truth, which may be ambiguous when LLMs depend on implicit context not reflected in the labels. Last, LLM predictions are non-deterministic and influenced by prior knowledge, which means the results may vary across different queries. Although CRYPTBARA issues each query five times and applies a majority vote to determine the final decision, this may still be insufficient in eliminating uncertainty.

VI. Related work

1) Python cryptographic API misuse detection: Several existing approaches aim to detect cryptographic API misuse in Python code. Acar *et al.* [3] conducted a usability study on libraries such as `PyCrypto` and `M2Crypto`, highlighting that poor documentation and insecure defaults often led to misuse. Wickert *et al.* [7] proposed *LICMA*, a rule-based tool, and found that Python had fewer misuses than Java but more than C, with many hidden in dependencies. Frantz *et al.* [8] introduced Cryptolation, a static analysis tool that leverages variable inference to detect cryptographic misuses in Python, and presented *PyCryptoBench*, a benchmark dataset comprising 1,836 labeled files for evaluation. Guo *et al.* [9] developed *CryptoPyt*, a static taint analysis tool using a custom AST to detect 17 misuse types. Gorski *et al.* [22] proposed security-advice, which provides runtime warnings to improve secure coding practices. Although these studies have advanced solutions for identifying Python cryptographic API misuse, they have not sufficiently considered issues that arise in complex real-world Python code (*e.g.*, context fragmentation), have limited inter-procedural analysis capabilities, and could not effectively analyze the semantics of Python code (see Table I). In contrast, CRYPTBARA demonstrates its distinctiveness by systematically classifying and identifying dependencies necessary for cryptographic API misuse analysis and leveraging dependency-guided LLM semantic reasoning to understand code semantics, thereby successfully identifying cryptographic API misuse (see Section IV).

2) Detecting cryptographic API misuse in other languages: Several studies have explored cryptographic API misuse detection in other languages (*e.g.*, [10], [23]–[33]).Some approaches rely on slicing techniques to detect misuses [23]–[25]. Piccolboni *et al.* [10] proposed CryLogger, a dynamic tool that monitors API calls at runtime. Paletov *et al.* [26] introduced DiffCode, which mines misuse patterns from GitHub commits and defines 13 rules using DAG-based graph abstraction. CryptoGo [27] and Gopher [28] use static taint analysis, with the latter supporting 31 rules via slicing and symbolic execution. However, applying these techniques to Python necessitates addressing its dynamic semantics and establishing language-specific rules. Thus, without substantial modifications, their effectiveness in Python environments remains limited.

3) Detecting general API misuses: Many studies have addressed API misuse across various domains (*e.g.*, [34]–[41]). However, these approaches do not address the unique challenges posed by Python's dynamic nature, particularly in the context of cryptographic APIs. Moreover, software composition analysis (*e.g.*, [42]–[46]) can identify vulnerable versions of cryptographic libraries; however, although these are effective at detecting the use of vulnerable APIs, they cannot be applied to identifying API misuses.

VII. Conclusion

In this paper, we presented CRYPTBARA, a hybrid detection framework that combines static dependency analysis with prompt-based LLM reasoning to precisely detect Python cryptographic API misuses. Our experiments demonstrate that CRYPTBARA outperforms state-of-the-art tools. It further uncovered 172 previously unknown misuses, with 22 critical cases reported to and confirmed by developers. Using CRYPTBARA, developers can identify potential misuses of cryptographic APIs in their Python code, ultimately improving the security and reliability of their software systems.

Data availability

The source code and dataset of CRYPTBARA are available at https://github.com/choseogyeong/CRYPTBARA.

Acknowledgment

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2022-II220277, Development of SBOM Technologies for Securing Software Supply Chains, and No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2025-00517788, Research on Intelligent SBOM Generation and Automated Vulnerability Analysis through Multi-level Code Analysis), and the Culture, Sports and Tourism R&D Program through the Korea Creative Content Agency grant funded by the Ministry of Culture, Sports and Tourism (International Collaborative Research and Global Talent Development for the Development of Copyright Management and Protection Technologies for Generative AI, RS-2024-00345025).

References

[1] M. Green and M. Smith, "Developers are Not the Enemy!: The Need for Usable Security APIs," *IEEE Security & Privacy*, vol. 14, no. 5, pp. 40–46, 2016.

[2] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You Get Where You're Looking For: The Impact of Information Sources on Code Security," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016, pp. 289–305.

[3] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the Usability of Cryptographic APIs," in *2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017, pp. 154–171.

[4] N. Meng, S. Nagy, D. D. Yao, W. Zhuang, and G. A. Argoty, "Secure Coding Practices in Java: Challenges and Vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. New York, NY, USA: ACM, 2018, pp. 372–383.

[5] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, and Y. Brun, "API Blindspots: Why Experienced Developers Write Vulnerable Code," in *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, 2018, pp. 315–328.

[6] M. A. Rahman, M. S. Hossain, and M. S. Uddin, "Cryptography in Python: A comprehensive overview and implementation," *IEEE Access*, vol. 10, pp. 123 456–123 470, 2022.

[7] A.-K. Wickert, L. Baumgärtner, F. Breitfelder, and M. Mezini, "Python Crypto Misuses in the Wild," in *Proc. 15th Int. Symp. on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2021, pp. 31:1–31:6.

[8] M. Frantz, Y. Xiao, T. S. Pias, N. Meng, and D. Yao, "Methods and Benchmark for Detecting Cryptographic API Misuses in Python," *IEEE Transactions on Software Engineering*, vol. 50, no. 5, pp. 1118–1129, 2024.

[9] X. Guo, S. Jia, J. Lin, Y. Ma, F. Zheng, G. Li, B. Xu, Y. Cheng, and K. Ji, "CryptoPyt: Unraveling Python Cryptographic APIs Misuse with Precise Static Taint Analysis," in *2024 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2024, pp. 1075–1091.

[10] L. Piccolboni, G. Di Guglielmo, L. P. Carloni, and S. Sethumadhavan, "CRYLOGGER: Detecting Crypto Misuses Dynamically," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021, pp. 1972–1989.

[11] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014, pp. 590–604.

[12] OWASP Foundation, "OWASP Application Security Verification Standard 4.0.3," Online, 2021, available at: https://github.com/OWASP/ASVS.

[13] ——, "OWASP Password Storage Cheat Sheet," Online, 2021, available at: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

[14] National Institute of Standards and Technology, "NIST Special Publication 800-133 Revision 2: Recommendation for Cryptographic Key Generation," Online, 2020, available at: https://doi.org/10.6028/NIST.SP.800-133r2.

[15] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "GPT-4 Technical Report," *arXiv preprint arXiv:2303.08774*, 2023.

[16] P. Liu, J. Liu, L. Fu, K. Lu, Y. Xia, X. Zhang, W. Chen, H. Weng, S. Ji, and W. Wang, "Exploring ChatGPT's Capabilities on Vulnerability Management," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 811–828.

[17] S. Woo, E. Choi, and H. Lee, "A large-scale analysis of the effectiveness of publicly reported security patches," *Computers & Security*, p. 104181, 2024.

[18] S. Woo, H. Hong, E. Choi, and H. Lee, "MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components," in *Proceedings of the 31st USENIX Security Symposium (Security)*, 2022, pp. 3037–3053.

[19] S. Woo, E. Choi, H. Lee, and H. Oh, "V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6541–6556.

[20] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017, pp. 595–614.

[21] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "LLaMA: Open and Efficient Foundation Language Models," *arXiv preprint arXiv:2302.13971*, 2023.

[22] P. L. Gorski, L. L. Iacono, D. Wermke, C. Stransky, S. Möller, Y. Acar, and S. Fahl, "Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic {API} Misuse," in *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, 2018, pp. 265–281.

[23] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 73–84.

[24] I. Muslukhov, Y. Boshmaf, and K. Beznosov, "Source Attribution of Cryptographic API Misuse in Android Applications," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 133–146.

[25] S. Rahaman, H. R. Nguyen, and T. N. Nguyen, "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*. ACM, 2019, pp. 2455–2472.

[26] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev, "Inferring Crypto API Rules from Code Changes," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2018, pp. 456–470.

[27] W. Li, S. Jia, L. Liu, F. Zheng, Y. Ma, and J. Lin, "CryptoGo: Automatic Detection of Go Cryptographic API Misuses," in *Annual Computer Security Applications Conference (ACSAC)*, 2022.

[28] Y. Zhang, B. Li, J. Lin, L. Li, J. Bai, S. Jia, and Q. Wu, "Gopher: High-Precision and Deep-Dive Detection of Cryptographic API Misuse in the Go Ecosystem," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[29] S. Krüger, K. Ali, E. Bodden, and M. Mezini, "CogniCrypt: Supporting Developers in Using Cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 931–936.

[30] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2341–2358, 2019.

[31] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. Yao, and N. Meng, "Automatic Detection of Java Cryptographic API Misuses: Are We There Yet?" *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 288–303, 2022.

[32] A. S. Ami, N. Cooper, K. Kafle, K. Moran, D. Poshyvanyk, and A. Nadkarni, "Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022, pp. 614–631.

[33] B. He, V. Rastogi, V. Balakrishnan, L. Ying, and W. Enck, "Vetting SSL Usage in Applications with SSLINT," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015, pp. 519–534.

[34] S. Amann, S. Amann, S. Nadi, and M. Mezini, "MUBench: A Benchmark for API-Misuse Detectors," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. ACM, 2016, pp. 464–467.

[35] X. Li, J. Jiang, S. Benton, Y. Xiong, and L. Zhang, "A Large-scale Study on API Misuses in the Wild," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2021.

[36] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, "Are Machine Learning Cloud APIs Used Correctly?" in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021.

[37] M. Wei, N. S. Harzevili, Y. Huang, J. Yang, J. Wang, and S. Wang, "Demystifying and Detecting Misuses of Deep Learning APIs," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2024.

[38] I. Yun, C. Lee, X. Wang, T. Kim, and M. Naik, "APISan: Sanitizing API Usages through Semantic Cross-checking," in *25th USENIX Security Symposium*. USENIX Association, 2016, pp. 363–378.

[39] Z. Gu, J. Wu, J. Liu, M. Zhou, and M. Gu, "An Empirical Study on Api-Misuse Bugs in Open-Source C Programs," in *2019 IEEE 43rd annual*

computer software and applications conference (COMPSAC), vol. 1. IEEE, 2019, pp. 11–20.

[40] Z. Li, A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song, "ARBITRAR: User-Guided API Misuse Detection," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[41] S. Bae, H. Cho, I. Lim, and S. Ryu, "SAFEWAPI: Web API Misuse Detector for Web Applications," in *Proceedings of the 2014 ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2014, pp. 1–11.

[42] J. Mahon, C. Hou, and Z. Yao, "PyPitfall: Dependency Chaos and Software Supply Chain Vulnerabilities in Python," *arXiv preprint arXiv:2507.18075*, 2025.

[43] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 860–872.

[44] Y. Na, S. Woo, J. Lee, and H. Lee, "CNEPS: A Precise Approach for Examining Dependencies Among Third-Party C/C++ Open-Source Components," in *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 2918–2929.

[45] Y. Choi and S. Woo, "TIVER: Identifying Adaptive Versions of C/C++ Third-Party Open-Source Components Using a Code Clustering Technique," in *Proceedings of the 47th International Conference on Software Engineering (ICSE). IEEE*, 2025.

[46] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying Open-Source License Violation and 1-day Security Risk at Large Scale," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2169–2185.