"MCP Does Not Stand for Misuse Cryptography Protocol": Uncovering Cryptographic Misuse in Model Context Protocol at Scale

BIWEI YAN, Shandong University, China
YUE ZHANG, Shandong University, China
MINGHUI XU, Shandong University, China
HAO WU, Nanjing University, China
YECHAO ZHANG, Shandong University, China
KUN LI, Shandong University, China
GUOMING ZHANG, Shandong University, China
XIUZHEN CHENG, Shandong University, China

The Model Context Protocol (MCP) is rapidly emerging as the middleware for LLM-based applications, offering a standardized interface for tool integration. However, its built-in security mechanisms are minimal: while schemas and declarations prevent malformed requests, MCP provides no guarantees of authenticity or confidentiality, forcing developers to implement cryptography themselves. Such ad hoc practices are historically prone to misuse, and within MCP they threaten sensitive data and services. We present MICRYSCOPE, the first domain-specific framework for detecting cryptographic misuses in MCP implementations. MICRYSCOPE combines three key innovations: a cross-language intermediate representation that normalizes cryptographic APIs across diverse ecosystems, a hybrid dependency analysis that uncovers explicit and implicit function relationships (including insecure runtime compositions orchestrated by LLMs) and a taint-based misuse detector that tracks sensitive data flows and flags violations of established cryptographic rules. Applying MICRYSCOPE to 9,403 MCP servers, we identified 720 with cryptographic logic, of which 19.7% exhibited misuses. These flaws are concentrated in certain markets (e.g., Smithery Registry with 42% insecure servers), languages (Python at 34% misuse rate), and categories (Developer Tools and Data Science & ML accounting for over 50% of all misuses). Case studies reveal real-world consequences, including leaked API keys, insecure DES/ECB tools, and MD5-based authentication bypasses. Our study establishes the first ecosystem-wide view of cryptographic misuse in MCP and provides both tools and insights to strengthen the security foundations of this rapidly growing protocol.

ACM Reference Format:
Biwei Yan, Yue Zhang, Minghui Xu, Hao Wu, Yechao Zhang, Kun Li, Guoming Zhang, and Xiuzhen Cheng. 2025. "MCP Does Not Stand for Misuse Cryptography Protocol": Uncovering Cryptographic Misuse in Model Context Protocol at Scale. 1, 1 (December 2025), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Biwei Yan, Shandong University, China, bwyan@sdu.edu.cn; Yue Zhang, Shandong University, China, zyueinfosec@sdu.edu.cn; Minghui Xu, Shandong University, China, mhxu@sdu.edu.cn; Hao Wu, Nanjing University, China, hao.wu@nju.edu.cn; Yechao Zhang, Shandong University, China, yech.zhang@gmail.com; Kun Li, Shandong University, China, kunli@sdu.edu.cn; Guoming Zhang, Shandong University, China, guomingzhang@sdu.edu.cn; Xiuzhen Cheng, Shandong University, China, xzcheng@sdu.edu.cn.

XXXX-XXXX/2025/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The rapid adoption of Model Context Protocol (MCP)[1] has reshaped how large language models (LLMs) interact with external tools, databases, and services. By decoupling model reasoning from tool execution, MCP provides a standardized interface for heterogeneous capabilities, enabling structured requests that are mediated by clients and executed by servers. This abstraction simplifies integration and creates a uniform communication channel, positioning MCP as a foundational layer in AI ecosystems. Recent developments further underscore its momentum: Microsoft has also positioned MCP as the "USB-C for AI apps," embedding it into its Windows AI Foundry to enable cross-platform interoperability [31]. Industry analyses suggest that MCP is on track to become the de facto middleware for intelligent systems, akin to HTTP in the web era, with forecasts indicating that more than 75% of enterprises will invest MCP by 2025 [28]. These trends highlight not only MCP's accelerating adoption [3, 6] but also its growing role as critical infrastructure in the agentic AI ecosystem.

Despite its promise, MCP's built-in security mechanisms remain minimal. At the protocol level, it enforces JSON-based schemas, capability declarations, and identifier traceability. While these safeguards help prevent malformed requests and enable basic auditing, they fall short in two critical areas: authenticity and confidentiality. MCP cannot natively guarantee that request–response messages are genuine or protected from interception, forcing developers to implement custom cryptographic safeguards such as encryption, authentication codes, or digital signatures.

Here lies the crux of the problem. History shows that when security-critical tasks are delegated to individual developers, cryptographic misuse is not the exception but the norm [2, 10, 14, 15, 20]. From hard-coded keys [5, 25] and fixed random seeds to weak hashes like MD5 [30, 34] and insecure modes such as ECB [4], decades of research have consistently demonstrated the prevalence of errors when developers directly handle cryptographic APIs. Within MCP, the risk is magnified: servers often mediate access to sensitive data and proprietary services, meaning that even subtle misuses such as predictable random values or unauthenticated encryption can escalate into systemic vulnerabilities with far-reaching consequences.

Unfortunately, existing program analysis and misuse detection tools are ill-suited to the unique challenges of MCP. First, MCP servers are highly heterogeneous, spanning over ten programming languages (e.g., Python, C++, Java, JavaScript) with differing type systems, memory models, and cryptographic libraries. Second, MCP tools are often weakly coupled, leaving it to the LLM at runtime to orchestrate function compositions that may combine otherwise benign operations into insecure workflows. For instance, a developer may provide one function that derives a key by simply truncating a password and another that encrypts data using AES-CBC. Individually, both functions appear reasonable (but when the LLM chains them together in response to a user prompt like "*encrypt this file with my password,*" the result is a dangerously weak encryption scheme). Third, even when the cryptographic APIs themselves are recognized, the real challenge lies in understanding intent. The same MD5 function may be perfectly acceptable when checking for accidental file corruption, yet dangerously inadequate when used for password storage. In other words, detecting misuse in MCP requires reasoning about semantics and context, not just pattern matching.

To address these challenges, we present MICRYSCOPE, a domain-specific analysis framework for detecting cryptographic misuses in MCP implementations. MICRYSCOPE introduces three key innovations: (i) a cross-language intermediate representation (IR) that normalizes cryptographic API usage across diverse ecosystems, ensuring consistent detection across languages; (ii) a hybrid dependency analysis that reconstructs both explicit and implicit relationships among functions in MCP's plugin-style architecture. Unlike traditional control-flow analysis, this approach models *may-dependencies* created at runtime when the LLM orchestrates weakly coupled functions, thereby exposing insecure compositions that static analysis alone cannot reveal; and (iii) a taint-based misuse

Fig. 1. LLM generates requests.

```
user_query = "What are the top 3 students by credits?"

if "credits" in user_query.lower():
    tool_spec = {
        "tool": "sql.query",
        "params": {
            "sql": "SELECT name, credits FROM students "
                   "ORDER BY credits"
                   "DESC LIMIT 3"
        }
    }
else:
    tool_spec = None
```

Fig. 2. Clients translate requests.

```
import uuid, requests
SERVER = "http://127.0.0.1:8000/mcp"
msg = {
    "id": str(uuid.uuid4()),
    "version": "0.1",
    "type": "invoke",
    "tool": "sql.query",
    "params": {"sql": "SELECT name, credits "
                      "FROM students ORDER BY credits DESC LIMIT 3"}
}
resp = requests.post(SERVER, json=msg).json()
```

Fig. 3. MCP Servers execute requests.

```
from fastapi import FastAPI, Request
import sqlite3

app = FastAPI()

@app.post("/mcp")
async def mcp_endpoint(req: Request):
    data = await req.json()
    sql  = data["params"]["sql"]
    con  = sqlite3.connect("students.db")
    cur  = con.cursor()
    cur.execute(sql)
    rows = cur.fetchall()
    return {"id": data["id"], "ok": True,
            "result": {"rows": rows}}
```

detector that tracks data flows across function boundaries and flags violations of well-established cryptographic security rules.

Through a large-scale study of 9,403 MCP servers, MICRYSCOPE identified 720 servers that implemented cryptographic logic, of which 19.7% exhibited misuses. These vulnerabilities are not evenly distributed: the Market Smithery Registry shows the highest density of insecure servers (42%), while Mcpmarket, though the largest platform, contains a lower proportion of misuses (37%). At the language level, Python servers (34%) are disproportionately prone to errors compared to other languages, reflecting the uneven quality of its cryptographic library ecosystem. From a functional perspective, Developer Tools and Data Science & ML categories account for more than 50% of all misuses, underscoring that critical flaws emerge in the very tools developers depend on most. Beyond statistics, our case studies highlight tangible consequences: hard-coded LLM API keys that attackers could immediately exploit for financial abuse, DES in ECB mode exposed through MCP tools that propagate insecure primitives into downstream applications, and MD5-based authentication tokens that allow adversaries to bypass deployment pipelines. "MCP" now being read as "*Misuse Cryptography Protocol*" rather than "*Model Context Protocol*".

In summary, this work makes the following contributions:

- We provide the first systematic study of cryptographic misuse in MCP, highlighting its causes, patterns, and consequences.
- We design MICRYSCOPE, a scalable analysis framework that integrates a cross-language intermediate representation, hybrid dependency reasoning, and taint-based detection to identify MCP-specific cryptographic misuses that existing tools fail to capture.
- We analyze 9,403 MCP servers and find that 19.7% of crypto-enabled servers contain misuses, mapping their prevalence across markets, languages, and categories, and demonstrating real-world consequences through case studies that expose financial, confidentiality, and integrity risks.

2 BACKGROUND

2.1 Model Context Protocol Workflow

Model Context Protocol (MCP) has been introduced as a standardized interface for enabling large language models to interact with external tools, data sources, and services in a secure and uniform manner. By decoupling model reasoning from tool execution, MCP establishes a context-sharing mechanism that allows models to invoke heterogeneous capabilities without relying on ad-hoc integration. The key components and their interactions can be summarized as follows:

- **LLM**: The large language model receives the user's input (e.g., the user may ask "*What are the top 3 students by credits?*") and first attempts to generate a response internally. When the task requires knowledge or capabilities beyond its training data (such as retrieving up-to-date information or executing a database lookup), the LLM produces a structured request and hands it off to the MCP client. Figure 1 shows how the LLM decides to issue a standardized request to the `sql.query` tool (In MCP, a tool is an externally exposed capability such as database query identified by a unique name and a defined input/output schema that the LLM can invoke through the client), instead of producing an answer solely from its internal knowledge.
- **MCP Client**: The client serves as the mediator between the LLM and the broader MCP ecosystem. It translates the LLM's structured request into the MCP protocol format, attaches metadata such as request identifiers and versioning, and sends it to the designated MCP server. This process ensures that the LLM never directly interacts with heterogeneous external systems. Instead, the client enforces consistency and abstraction, so the model communicates through a uniform channel regardless of the underlying resource. Figure 2 illustrates how the client *constructs the JSON request* (i.e., *"query the student table for the top 3 by credits"*) and POSTs it to the server. This framing ensures the LLM never touches the database interface directly and that downstream components can apply uniform validation and logging.
- **MCP Server**: The server provides controlled access to external resources. Upon receiving a request from the client, it validates the message, enforces policy restrictions (e.g., read-only SQL), executes the operation on the outside resource, and returns results in the standardized MCP format. For instance, an MCP server may connect to a SQL database, call a *REST API*, or query a proprietary dataset. By encapsulating these details, the server hides complexity from both the client and the LLM, ensuring modularity and extensibility. Figure 3 presents a compact handler for `/mcp` that (i) reads the `sql` parameter, (ii) executes it on the "student" table, and (iii) returns a standardized JSON `result`. This result then flows back through the MCP client to the LLM, which integrates the retrieved rows into its generated response and presents the final natural-language answer to the user.

Please note that the end user interacts directly with the LLM through a conversational or application interface. The user formulates a query, provides context, or issues a command. Importantly, the user does not need to know which external resources or tools may be involved. From the user's perspective, the LLM acts as the single entry point for intelligent reasoning and task execution.

2.2 MCP Security and Limitations

MCP provides several intrinsic security properties at the protocol level. These constitute a minimal security baseline: First, MCP enforces strict compliance with a JSON-based request–response schema that is compatible with JSON-RPC. All messages must conform to predefined structural rules, which reduces the risk of arbitrary injection or malformed input. Second, at session initialization, the MCP server must explicitly declare the tools and operations it supports, including the expected parameter structures. This handshake ensures that no undeclared or hidden functionalities can be invoked. In effect, this mechanism acts as a protocol-level whitelist, constraining interactions to a well-defined operational space. Finally, every MCP request carries a unique identifier that must be echoed in the corresponding response. This design enforces consistent pairing between request and response, prevents message confusion or replay within a session, and provides basic traceability across the communication channel.

However, the native safeguards provided by MCP are limited in scope and primarily oriented toward basic validation and traceability. MCP lacks intrinsic support for two fundamental security

dimensions: authenticity and confidentiality of request–response exchanges. Without these guarantees, the framework cannot inherently verify the legitimacy of a message's origin or ensure that its content has not been modified in transit. Likewise, the absence of built-in confidentiality protections leaves sensitive information exchanged between clients and servers potentially exposed to interception or leakage. Consequently, many MCP implementations supplement the baseline framework with additional cryptographic mechanisms such as digital signatures, message authentication codes, and secure transport protocols to provide end-to-end authenticity and confidentiality guarantees.

3 PROBLEM STATEMENT AND CHALLENGES

3.1 Motivation and Problem Statement

As discussed in §2.2, while MCP incorporates basic safeguards such as input validation, capability declaration, and request traceability, it notably lacks built-in support for two fundamental security properties: request–response authenticity and confidentiality. As a result, developers of MCP servers frequently implement custom cryptographic mechanisms to provide these guarantees, for instance by incorporating encryption, message authentication codes, or digital signatures at the application layer.

However, the reliance on developer-supplied cryptographic logic introduces a significant risk of misconfiguration and misuse. Prior studies of cryptographic libraries and APIs have consistently demonstrated that developers often select weak primitives, apply insecure modes of operation, or omit essential steps such as randomness generation and key validation. In the context of MCP, such misuses are particularly concerning: insecure implementations may allow adversaries to forge responses, tamper with sensitive requests, or exfiltrate confidential data through improperly protected channels. Given that MCP servers often act as trusted gateways to databases, proprietary APIs, and sensitive user information, the consequences of cryptographic misuse extend well beyond the protocol itself and can compromise the integrity of the broader system. Despite the prevalence of these risks, there is currently no systematic approach tailored to identifying cryptographic misuse within MCP implementations. Existing static and dynamic analysis tools are designed for general-purpose applications and lack the domain-specific awareness of MCP's message structures, lifecycle stages, and protocol-specific interaction patterns. *Our goal, therefore, is to design a domain-specific analysis tool that systematically detects and characterizes cryptographic misuses in MCP implementations, thereby providing practitioners with actionable insights to harden their deployments.*

3.2 Challenges

Detecting cryptographic misuse in the MCP ecosystem is far from straightforward. Unlike traditional software systems, MCP servers introduce unique complexities that fundamentally hinder the applicability of existing program analysis and security auditing tools. The challenges are not limited to simple engineering obstacles such as parsing multiple languages or instrumenting code; rather, they reflect deeper issues in modeling heterogeneous implementations, reasoning about dynamic execution semantics, and interpreting the security implications of cryptographic API usage.

C1. Multi-Language Heterogeneity in Implementations. Within the MCP ecosystem, server implementations are not confined to a single programming language. Owing to the protocol's openness and its demand for cross-platform deployment, MCP servers are typically realized in more than ten different languages, including Python, Java, C++, PHP, Swift, and Go. This heterogeneity enhances flexibility and broadens adoption, but it simultaneously creates formidable challenges for static analysis tasks such as cryptographic misuse detection.

Cryptographic misuse detection typically involves identifying crypto-related API invocations, tracing their data-flow dependencies, and validating whether parameters conform to established

security practices. Most existing tools are designed for a single language and lack the cross-language modeling capability required to analyze MCP's heterogeneous ecosystem. The difficulty lies not only in parsing distinct syntax and grammar but also in capturing subtle semantic differences between languages. As summarized in Table 1, implementations vary widely across several dimensions: typing discipline (e.g., strongly typed Java, Swift, and Go versus dynamically typed Python, PHP, and JavaScript), memory management (manual in C++ versus automatic in managed environments), concurrency models (traditional threads, event-driven loops, or goroutines), and the prevalence of implicit defaults (e.g., padding or mode assumptions in PHP and Node.js).

Table 1. Language Heterogeneity in MCP servers

Language	Library Example	Typing	Manual Memory	Async	Implicit Defaults
Java	`javax.crypto`	Strong	✗	✗	✗
Python	`PyCryptodome`	Weak	✗	✗	✓
C++	OpenSSL, Botan	Strong	✓	✗	✗
PHP	OpenSSL extension	Weak	✗	✗	✓
Swift	CommonCrypto	Strong	✗	✗	✗
JavaScript	`crypto` module	Weak	✗	✓	✓
Go	`crypto/aes`	Strong	✗	✓	✗

These discrepancies fundamentally affect how cryptographic APIs are invoked and how their security properties can be validated. For instance, detecting insecure use of AES/ECB mode in Java requires analyzing structured class hierarchies and checked exceptions, while the same misuse in Python is hidden within dynamically imported modules and aliased functions. Similarly, C++ demands precise reasoning over pointer semantics and API state machines, whereas PHP introduces risks through runtime configuration and implicit defaults. A single-language analyzer cannot seamlessly adapt to these variations, leaving significant blind spots in detecting misuse across the MCP ecosystem.

C2. Weakly Coupled Functions and Implicit Flows. The second challenge arises from the way functions are organized within individual MCP tools. Unlike conventional software modules, where functions are often explicitly linked through internal call relationships, MCP tools typically expose a set of weakly coupled functions without a predetermined orchestration order. At runtime, it is the LLM (guided by natural language prompts) that decides how to combine these functions to accomplish a task. While this plugin-style modularity provides flexibility and extensibility, it also obscures the true execution paths and creates opportunities for cryptographic misuses that are invisible to traditional static analysis.

To demonstrate this challenge, we provide a simplified example shown in Figure 4. Individually, these functions appear legitimate: `derive_key()` outputs a key-like string from a password, and `encrypt_cbc()` performs AES encryption. Yet, if a user issues a prompt such as *"encrypt my file with a password"*, the LLM may compose these functions by first deriving the key with `derive_key()` and then encrypting with `encrypt_cbc()`. The resulting execution path produces *AES encryption with a weakly derived key*, which is a severe misuse. Importantly,

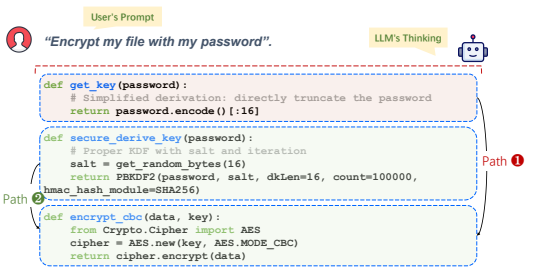


Fig. 4. An MCP tool with weakly coupled functions for key derivation and encryption

this flaw does not originate from a single function but emerges from the implicit composition of otherwise "reasonable" utilities. In contrast, the same instruction could also lead the LLM to select an alternative combination (e.g., a more secure `secure_derive_key()` function with a proper KDF and a `encrypt_cbc()` function with randomized IVs), resulting in a secure execution. The critical point is that these different execution paths are not encoded as explicit control-flow edges in the tool's code. Instead, they exist only as potential compositions resolved at runtime, outside the scope of conventional control-flow graph (CFG) construction.

This architectural property highlights why cryptographic misuse detection in MCP cannot rely solely on traditional static analysis. Whereas classical tools reason over explicitly defined function calls, in MCP the decisive factor is the LLM's runtime orchestration of weakly coupled functions. Misuses therefore arise not from isolated function implementations but from implicit execution paths that materialize only during prompt-driven interaction. Addressing this challenge requires analysis techniques that move beyond intra-function correctness and explicitly account for dynamic, runtime composition within tools.

C3: Detecting Misuse from Static Dependencies. Even after achieving cross-language modeling (C1) and accurate identification of data dependencies (C2), a further challenge lies in determining whether cryptographic APIs are being misused within MCP implementations. Cryptographic misuse is not always syntactically obvious: the same API may be legitimate in one context yet insecure in another. Distinguishing misuse thus requires semantic reasoning about the purpose of an operation and the surrounding security context, not just identifying the function call itself.

```python
import hashlib
def checksum(file_path):
    with open(file_path, "rb") as f:
        data = f.read()
    return hashlib.md5(data).hexdigest()
```

```python
import hashlib
def store_password(password):
    # Insecure: directly hash password with MD5
    return hashlib.md5(password.encode()).hexdigest()
```

Fig. 5. Contrasting uses of MD5: benign file checksums (left) vs. insecure password storage (right)

A classic example is the use of MD5. In some cases, MD5 is employed merely for file integrity checking, which may be benign; in others, it is applied in security-sensitive contexts such as password storage, which constitutes a critical misuse. The contrast is illustrated in Figure 5. As shown in the figure, the first function computes a checksum for detecting accidental file corruption. Although MD5 is broken in terms of collision resistance, such use may be acceptable where adversarial manipulation is not a concern. The second function, however, applies MD5 for password storage, a context where its weaknesses are catastrophic: lack of salting, key stretching, and reliance on a broken hash function make it trivially vulnerable to offline brute-force attacks.

These examples demonstrate that identifying an MD5 invocation alone is insufficient to judge security. Correct classification requires reasoning about intent (checksum vs password hashing), data type (arbitrary file vs sensitive credentials), and security requirements. In MCP, this difficulty is compounded by cross-language diversity, as the same API misuse may appear in Python (`hashlib.md5()`), Java (`MessageDigest.getInstance("MD5")`), or PHP (`openssl_digest("md5")`). Thus, the core challenge is to map observed API invocations to their security semantics and to decide whether they align with or violate best practices. Without this capability, tools risk either over-reporting benign uses or, worse, overlooking subtle yet dangerous misuses.

4 DESIGN OF MICRYSCOPE

We designed MICRYSCOPE (Misuse In CRYptography), which is designed to act as a "microscope" for MCP servers. We first outline the key solutions (§4.1) to the identified challenges and then present the detailed design (§4.2).

4.1 Key Solutions

(S1) Cross-Language Abstraction via Unified IR. The first challenge (C1) highlights that the heterogeneity of MCP server implementations across many programming languages makes it extremely difficult to build a unified analysis framework. To address this, we abstract away language-level idiosyncrasies by converting source code into abstract syntax trees (ASTs), which provide a structured yet language-agnostic view of program semantics. Building on ASTs, we systematically extract cryptography-related function calls, including their invocation sites, input parameters, return values, and surrounding context. For instance, we capture not only the literal arguments to a cryptographic API (e.g., key size, cipher mode) but also the symbolic variables whose values may flow into these calls. To enable cross-language analysis, we normalize all such information into a structured intermediate representation (IR) expressed in JSON. Each IR node encodes the API name, its lexical scope, parameter semantics, produced variables (such as keys, ciphertexts, or IVs), and the inferred data dependencies.

(S2) Dependencies Construction via Must/May Analysis. The second challenge (C2) highlights that the plugin-style architecture of MCP tools exposes functions as weakly coupled utilities without explicit invocation chains, which makes traditional control-flow analysis (e.g., CFGs) ineffective for recovering inter-function dependencies. To overcome this limitation, we adopt a two-level dependency reconstruction strategy. First, we perform *Def–Use Analysis*, which in our context serves as a form of *must analysis*. From the IR, we extract each function's input parameters and output variables, and construct a global interprocedural Def–Use graph. This graph encodes deterministic dependencies: whenever a variable definition can be unambiguously linked to a subsequent use, a must edge is inserted. In this way, the graph recovers explicit and verifiable data flows across functions that would otherwise remain disconnected.

However, MCP also involves numerous implicit dependencies that cannot be captured by must analysis alone. For example, two functions may appear independent in code, yet when orchestrated by the LLM in response to a prompt, they may operate on the same underlying resource (e.g., a file path, URL, or storage key). Such latent relationships escape Def–Use Analysis because no explicit variable passing occurs. To account for these cases, we introduce *May Analysis*. Specifically, we normalize IR parameters by abstracting resource identifiers (paths, URLs, bucket/key pairs) into canonical fingerprints, and by unifying variable references into common-name entities that reflect their ultimate source. Whenever multiple functions are

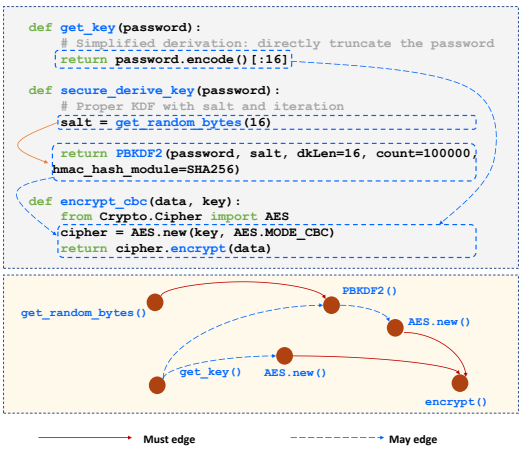


Fig. 6. Must vs. May Dependency in Key Derivation

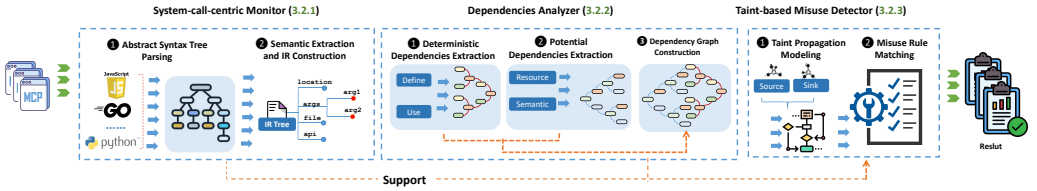


Fig. 7. Design of MICRYSCOPE

detected to access the same fingerprint or entity, we conservatively insert may edges to represent potential dependencies. By combining deterministic must edges from Def–Use Analysis with conservative may edges from resource-based normalization, we construct a comprehensive dependency graph. This hybrid graph not only recovers explicit data flows but also exposes hidden, prompt-driven dependencies between otherwise unrelated functions. As such, it provides a structured foundation for reasoning about execution order and for reliably identifying cryptographic misuse in MCP servers.

As shown in Figure 6, the use of `salt` inside `secure_derive_key` creates a *must edge*, since the variable is deterministically defined by `get_random_bytes(16)` and directly consumed by `PBKDF2`. This represents a concrete and unambiguous dependency. By contrast, the `key` variable introduces a *may edge*: depending on runtime orchestration, it may be derived insecurely via `get_key(password)` (simple truncation) or securely via `secure_derive_key(password)` (proper KDF with salt). Because this dependency cannot be conclusively resolved at the static level, it must be conservatively modeled as a potential relation.

(S3) Data Flow Tracking via Taint Analysis. Building on the intermediate representation (S1) and dependency graph constructed (S2), we introduce *Taint Analysis* as the core mechanism for detecting cryptographic misuse. This approach systematically tracks the propagation of potentially untrusted data through program execution, thereby uncovering misuse patterns that would otherwise remain hidden. Concretely, the analysis first identifies *taint sources*, such as user inputs, configuration files, or environment variables, which may inject untrusted data into the system. It then designates *taint sinks*, including cryptographic API calls, data output operations, and file writes, as sensitive nodes where misuse may manifest. Next, the taint analysis performs *propagation* along the dependency graph. Whenever a variable obtains data from a taint source, its taint status is preserved and propagated through subsequent assignments, function calls, and return values. This mechanism explicitly captures the flow of untrusted inputs into sensitive cryptographic operations, while retaining contextual awareness across function boundaries. Finally, the resulting propagation chains are checked against a set of *misuse detection rules* (See §4.2). These rules encode well-known insecure patterns, such as hard-coded keys, fixed initialization vectors, weak algorithms (e.g., MD5), or missing authentication steps. If a propagation path matches one of these patterns, the corresponding operation is flagged as a misuse.

4.2 Detailed Design

As shown in Figure 7, MICRYSCOPE introduces a cross-language intermediate representation to unify cryptographic API usage, a hybrid dependency analysis to recover both explicit and implicit function relationships in plugin-style architectures, and a taint-based misuse detector to trace data flows and enforce established cryptographic rules.

4.2.1 AST-Guided IR Extractor. To address **C1**, the MICRYSCOPE leverages the structured representation capability of ASTs to systematically parse MCP server source code and construct an IR suitable for cross-language cryptographic misuse detection.

(Step I) Abstract Syntax Tree Parsing. We first utilize ecosystem-specific parsers (e.g., Python's `ast`, JavaScript's `@babel/parser`) to convert source code into ASTs. To enrich their contextual sensitivity, we enhance the raw ASTs in two key ways: First, each node is assigned a reverse reference to its parent, enabling contextual awareness (e.g., detecting whether a function call occurs on the right-hand side of an assignment). Second, a scope stack and symbol table are maintained to track variable bindings and visibility, which provides the basis for subsequent Def–Use analysis (See §4.2.2).

(Step II) Semantic Extraction and IR Construction. On top of the enriched ASTs, the system traverses all function call nodes and extracts their key features, including call name, file location, scope, and parameters. Each call is then normalized into a language-independent IR unit with the following properties: (i) All calls are uniformly encapsulated into IR nodes, allowing for consistent analysis across different languages. (ii) Parameters are semantically tagged into categories such as constant, `list_literal`, `dict_literal`, `function_return`, and `variable`.

4.2.2 Dependencies Analyzer. We leverage the IR as the carrier to recover inter-function relationships in the plugin-style MCP architecture, where explicit call chains are often absent. As shown in algorithm 1, we design a two-level dependency extraction mechanism that combines *must-dependence* and *may-dependence* analysis, followed by graph construction.

(Step I) Deterministic Dependencies Extraction. We apply a lightweight Def–Use analysis over IR nodes. Each call node specifies both its produced variable (`produced_as`) and consumed arguments. When a variable defined in one call appears as a parameter of another call, the system matches the pair to form a deterministic Def → Use edge. For example, if the first call generates a random `salt` and stores it in variable `salt_val`, and the second call later passes `salt_val` as an argument to a key derivation API (e.g., `PBKDF2`), the framework records a must-dependence edge. By exhaustively scanning all call pairs within the file scope, the system constructs the complete set of explicit dependencies.

(Step II) Potential Dependencies Extraction. Explicit Def–Use relationships, however, are insufficient for weakly coupled modules common in MCP. To capture implicit yet semantically meaningful relationships, we introduce may-dependence edges through resource fingerprinting and API semantic categorization:

- **Resource fingerprinting** extracts path names, message topics, or identifiers from arguments of type `constant`, treating them as resource anchors across calls. For example, a `write_file` call that outputs data to `"user.db"` and a subsequent `read_file` call accessing the same path are linked via the constant `"user.db"` as a resource fingerprint, forming a may-dependence edge.
- **Semantic categorization** groups APIs into high-level functions such as `protect`, `upload`, or `mask`, enabling consistent analysis across languages. A may-dependence edge is introduced when two calls both operate on the same resource (via a shared fingerprint such as a filename or overlapping variable names such as "*key*") and their functions match a high-risk pattern. For example, consider a call `encrypt(data, key)` categorized as `protect`, followed by a call `upload("user.db")` categorized as `upload`. Although no variable is explicitly passed between them, both share the fingerprint `"user.db"`. The system therefore records a may-dependence edge `encrypt` → `upload`, modeling the implicit flow of sensitive data from local protection to external exposure.

Algorithm 1: Dependency Extraction

Input: IR set $\mathcal{I}$ of all call nodes from project P
Output: Dependency graph $G = (V, E)$ with must- and may-dependence edges

Initialize $V \leftarrow \mathcal{I}, E \leftarrow \emptyset$;
Step 1: Must-dependence extraction (Def–Use) ;
foreach *call node* $c \in \mathcal{I}$ **do**
  foreach *argument* $a \in c.arguments$ **do**
    if $a.type = variable$ **then**
      $v \leftarrow a.value$;
      foreach *call node* $d \in \mathcal{I}$ **do**
        if $d.produced\_as = v$ **then**
          add edge $(d \rightarrow c)$ to E_{must};
Step 2: May-dependence extraction ;
foreach *call node* $c \in \mathcal{I}$ **do**
  extract $fp(c)$ from arguments of type `constant` (path, topic, ID);
  assign semantic label $sem(c)$ from API categorization;
foreach *pair* (c_i, c_j) *with* $i \neq j$ **do**
  if $(sem(c_i), sem(c_j)) \in$ *risky patterns* **then**
    if $fp(c_i) = fp(c_j)$ ***or*** *shared variable in parameters* **then**
      add edge $(c_i \rightsquigarrow c_j)$ to E_{may};
Step 3: Graph construction;
$E \leftarrow E_{must} \cup E_{may}$;
return $G = (V, E)$;

(Step III) Dependency Graph Construction. Finally, both must- and may-dependence edges are integrated into a unified dependency graph. Each IR node is represented as a graph vertex, while edges denote either must or may relationships. The resulting graph is directed, typed, and layered: nodes encode API calls with their attributes, edges encode dependence semantics, and subgraphs naturally emerge for modules or plugins. This enriched dependency graph not only preserves precision from deterministic analysis but also broadens coverage by incorporating semantic cues, forming the structural foundation for downstream misuse detection.

4.2.3 Taint-based Misuse Detector. In the third component, we apply taint analysis to uncover insecure configurations and data leakage paths in cryptographic API usage. The process consists of two steps: taint propagation modeling and misuse rule matching.

Table 2. Eight Rules for Cryptographic API Misuse Detection

ID	Rule	Description
R1	Fixed Key / API Key	Hard-coded encryption or API keys directly embedded in code, making them easily recoverable and reusable by attackers [2, 4, 20, 23, 27, 32].
R2	Fixed IV / Salt	Use of constant IVs or salts, which weaken randomness and compromise security [2, 4, 23, 27, 32].
R3	Weak Hash Functions	Usage of MD5, SHA1, or other broken hash algorithms in security-sensitive contexts [2, 4, 23, 30, 35].
R4	Insecure Key Derivation Configuration	Use of password-based encryption (PBE) with insufficient iterations or weak parameter settings [2, 4, 20].
R5	Static Seed in PRNG	PRNG initialized with a fixed seed, producing deterministic random sequences [2, 4, 23, 27].
R6	ECB Mode Usage	Use of ECB block cipher mode, which leaks plaintext structure through repeated patterns [2, 4, 20, 23, 32].
R7	Missing Integrity Protection	Encryption applied without authentication (e.g., AES without MAC/GCM), allowing undetected tampering [2, 4].
R8	Deprecated Alg/APIs	Use of outdated cryptographic primitives (e.g., DES, RC4) or unsafe APIs [2, 4, 23].

(Step I) Taint Propagation Modeling. Over the constructed dependency graph, each argument and produced variable is semantically annotated (e.g., `constant`, `variable`, `function_return`, `dict_literal`, `list_literal`). By recursively tracing the Def–Use chains, the system resolves intermediate variables to their concrete sources (e.g., literals or fixed configurations), thereby exposing hidden hard-coded secrets or unsafe defaults. Taint analysis is then performed: external inputs are defined as *sources*, while operations such as printing, persistence, or network transmission are defined as *sinks*. A forward traversal of the graph is conducted to determine not only whether a flow exists but also *how it flows*. To systematically capture common pitfalls, we summarize ten representative rules of cryptographic API misuse in Table 2. Specifically, the analysis evaluates whether a propagated key is hard-coded, whether an IV is constant, whether a weak hash function is used, or whether randomness is seeded deterministically. The analysis spans multiple functions and files while maintaining consistent interprocedural context.

(Step II) Misuse Rule Matching. Once taint propagation identifies insecure propagation chains, the system applies misuse detection rules to flag concrete vulnerabilities. A project is marked as misusing cryptography if a tainted artifact reaches a sink under insecure conditions, such as constant keys, fixed IVs, weak hash algorithms, or predictable random seeds. The combination of dependency graph analysis and taint semantics thus enables systematic and fine-grained detection of cryptographic misuses across heterogeneous MCP implementations.

5 EVALUATION

5.1 Experiment Setup

Data Collection. We systematically collected MCP server projects from GitHub API and multiple external registries, including *Smithery, Pulse MCP, Cursor Directory, Awesome MCP, Glama AI, Mcpmarket*, and *Modelcontextprotocol.io.* A customized crawler was implemented to automatically retrieve both the source code and the corresponding metadata of MCP servers. In total, we obtained 9,403 MCP servers, covering a wide range of functional categories. Among them, Developer Tools dominate the dataset, with more than 2,300 instances from *Mcpmarket* alone. To mitigate class imbalance and avoid bias from sparsely distributed categories, we grouped the remaining servers into an "*Other*" category, ensuring a comprehensive and representative dataset. In addition, a small portion of servers is labeled "*Unknown*", reflecting incomplete or inconsistent metadata in public registries.

As shown in Figure 8, our analysis of those servers reveals several noteworthy patterns in markets and categories distribution. First, the ecosystem demonstrates a concentrated yet multipolar structure: while Mcpmarket alone accounts for roughly one quarter of all servers, Smithery Registry and Pulse MCP also contribute substantial shares, together forming a small set of dominant platforms. Second, in terms of functionality, Developer Tools overwhelmingly dominate the landscape, with more than half of all servers (~5,000) falling into this category, underscoring the developer-centric nature of the MCP ecosystem. Finally, although secondary categories such as Data Science & ML, Database Management, and Web Scraping

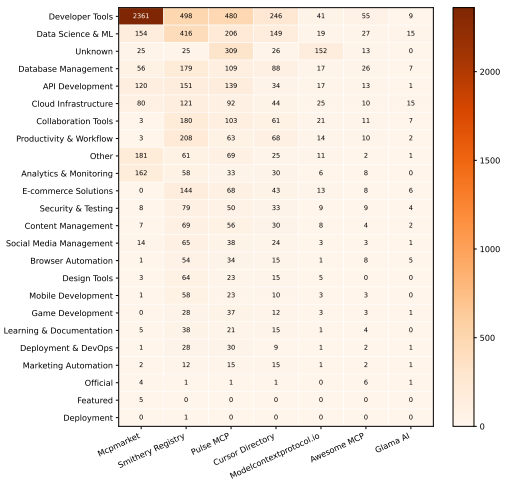


Fig. 8. Distribution of MCP Servers

are represented, application-oriented domains such as e-commerce, social media, and content management remain marginal, suggesting that MCP adoption beyond developer-focused use cases is still in its early stages.

Execution Environment. All experiments were conducted on a workstation equipped with an Intel(R) Core(TM) Ultra 7 258V @ 2.20 GHz processor, 32 GB RAM, and Windows 11 Pro (Version 24H2, OS Build 26100.4946). The system runs on a 64-bit x64-based architecture. This configuration offers sufficient computational power for large-scale parsing, IR construction, and cryptographic misuse detection.

Implementation. Our analysis pipeline was primarily implemented in Python (3.11) with approximately 6,000 lines of code. Since extracting ASTs required handling language-specific features, we employed parsers in multiple languages (e.g., JavaScript and TypeScript for front-end plugins, Python for backend servers) to generate ASTs.

5.2 Performance of MICRYSCOPE

Time Overhead. We evaluate the end-to-end performance of MICRYSCOPE when applied to the entire dataset of 9,403 MCP servers. As shown in Figure 9, the total execution time was broken down into three major stages: IR generation, dependency extraction, and misuse detection. The overall runtime is dominated by IR generation, which accounts for more than half of the total analysis time. This is expected, as multi-language AST parsing and normalization into the unified IR format is the most resource-intensive step. In contrast, dependency extraction (must-/may-dependence analysis) and misuse detection (taint propagation and rule matching) together contribute a smaller fraction of the total runtime, showing that the heavy lifting lies in IR construction. The distribution suggests no sharp outliers: runtime grows smoothly with dataset size, and the full corpus can be processed in a feasible timeframe on a single workstation. This indicates that MICRYSCOPE is capable of handling large-scale MCP ecosystems and can be readily extended to even larger registries as they emerge.

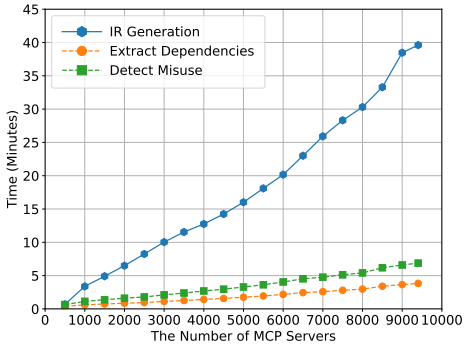


Fig. 9. Performance of our MICRYSCOPE

Accuracy. MICRYSCOPE demonstrates high accuracy. In manual validation, 5 of 100 randomly sampled servers classified as misuse-free were found to contain misuses (5% FN), while all 142 flagged cases were confirmed as true misuses (0% FP). These cases largely arise from two limitations. First, MICRYSCOPE relies on static IR construction and taint propagation. When cryptographic parameters are generated dynamically at runtime (e.g., IVs derived from system time or environment randomness), they may remain as unresolved variables in the IR rather than being reduced to insecure constants, causing misuses to be overlooked. Second, some projects embed cryptographic logic indirectly through wrappers, utility functions, or domain-specific libraries. If such patterns fall outside MICRYSCOPE 's rule set, they may evade detection. One representative case is the *MongoDB_Atlas* server (Figure 10), where a custom `md5` wrapper internally invoked the standard library. Because

sensitive values such as API keys, realms, and nonces were concatenated into compound strings before hashing, their roles were obscured, and the weak MD5 usage was missed. This combination of API hiding and string manipulation demonstrates how indirect or obfuscated implementations can reduce static visibility and lead to false negatives.

```
private md5(data: string): string {
  return crypto.createHash('md5').update(data).digest('hex');}
const ha1 = this.md5(`${this.apiKey}:${authDetails.realm}:${this.privateKey}`);
const ha2 = this.md5(`${method}:${new URL(url).pathname}`);
const response = this.md5(`${ha1}:${authDetails.nonce}:${nc}:${cnonce}:${authDetails.qop}:${ha2}`);
```

Fig. 10. False negative example where MICRYSCOPE missed weak MD5 usage due to custom wrapper and string concatenation

Table 3. Language Distribution of Crypto Adoption and Misuse Across MCP Markets

Language	Mcpmarket				Smithery Registry				Pulse MCP				Cursor Directory				MCP.io				Awesome MCP				Glama AI			
	Crypto		Misuse		Crypto		Misuse		Crypto		Misuse		Crypto		Misuse		Crypto		Misuse		Crypto		Misuse		Crypto		Misuse	
	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗
Javascript	112	1555	17	1650	142	1390	24	1508	143	1614	12	1745	30	535	3	562	31	287	5	313	14	125	1	138	2	46	-	48
Python	89	1152	30	1211	91	813	32	872	-	-	-	-	18	355	4	369	-	-	-	-	3	69	-	72	1	30	-	31
Go	11	131	4	138	6	53	3	56	5	108	1	112	-	22	-	22	4	26	1	29	-	11	-	11	-	-	-	-
Java	2	47	2	47	4	24	-	28	1	38	1	38	-	12	-	12	-	8	-	8	-	-	-	-	-	-	-	-
Rust	-	37	-	37	1	6	1	6	-	38	-	38	-	6	-	6	-	10	-	10	-	1	-	1	-	-	-	-
TypeScript	2	13	-	15	-	-	-	-	2	23	-	25	3	6	1	8	-	4	-	4	-	1	-	1	-	-	-	-
C#	-	26	-	26	-	5	-	5	-	9	-	9	-	2	-	2	-	-	-	-	-	2	-	2	-	-	-	-
Swift	-	9	-	9	-	-	-	-	-	9	-	9	-	3	-	3	-	1	-	1	-	1	-	1	-	-	-	-
Ruby	-	7	-	7	-	1	-	1	-	5	-	5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PHP	-	3	-	3	1	1	-	2	2	2	-	4	-	1	-	1	-	-	-	-	-	-	-	-	-	-	-	-

5.3 Empirical Results

In total, we identified 720 MCP (out of 9,403) servers that contained cryptographic operations, among which 142 instances (19.7%) exhibited cryptographic misuses. To better understand the distribution and characteristics of these misuses, we conduct a detailed analysis from three complementary perspectives: (i) the distribution across different markets, (ii) the programming languages employed, (iii) the prevalence within functional categories, and (iv) the breakdown with respect to specific misuse rules.

(I) Market-Level Analysis of Misuse. We begin by examining the market-level distribution of misuses across the seven major MCP registries. It can be observed from Figure 11 that while Mcpmarket dominates the ecosystem overall (3,196 servers, more than double the next-largest Smithery Registry with 2,538), its share of misuses (37%) is slightly lower than Smithery's (42%). This indicates that Smithery Registry, despite being smaller in absolute size, harbors a disproportionately high number of insecure implementations. Pulse MCP, with 1,999 servers, contributes 10% of the misuses, while Cursor Directory and Modelcontextprotocol.io show smaller absolute counts, but their relative misuse rates are not negligible given their more modest project bases. Finally, Awesome MCP account for only 1% of misuses, yet their presence highlights that even niche or community-driven platforms are not immune to insecure cryptographic practices. Although the ecosystem is highly centralized

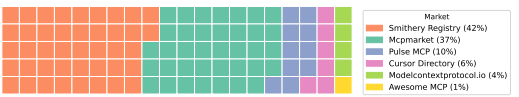


Fig. 11. MCP Servers with Crypto Misuse by Market

around Mcpmarket, the greatest density of misuses lies in Smithery Registry, suggesting that misuse is influenced not just by market size, but also by the curation standards and developer practices characteristic of each registry.

Table 4. Cross-Market Category Distribution with Crypto Adoption and Misuse

Language	Mcpmarket				Smithery Registry				Pulse MCP				Cursor Directory				MCP.io				Awesome MCP				Glama AI			
	Crypto		Misuse		Crypto		Misuse		Crypto		Misuse		Crypto		Misuse		Crypto		Misuse		Crypto		Misuse		Crypto		Misuse	
	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗
Developer Tools	161	2200	38	2323	37	461	3	495	31	449	1	479	10	236	1	245	3	38	1	40	1	54	-	55	1	8	-	9
Data Science & ML	6	148	3	151	47	369	17	399	16	190	2	204	8	141	2	147	2	17	-	19	1	26	-	27	-	15	-	15
Database Management	4	52	-	56	16	163	1	178	6	103	-	109	6	82	1	87	1	16	-	17	4	22	-	26	-	7	-	7
Collaboration Tools	1	2	-	3	14	166	5	175	7	96	2	101	1	60	1	60	-	21	-	21	-	11	-	11	-	7	-	7
Cloud Infrastructure	5	75	-	80	15	106	3	118	7	85	-	92	7	37	1	43	5	20	2	23	2	8	-	10	1	14	-	15
Productivity & Workflow	-	3	-	3	26	182	9	199	7	56	-	63	4	64	1	67	-	14	-	14	3	7	1	9	-	2	-	2
Analytics & Monitoring	8	154	3	159	2	56	-	58	5	28	-	33	-	30	-	30	1	5	-	6	-	8	-	8	-	-	-	-
E-commerce Solutions	-	-	-	-	14	130	5	139	5	63	-	68	2	41	-	43	-	13	-	13	1	7	-	8	1	5	-	6
Security & Testing	3	5	1	7	11	68	3	76	5	45	2	48	2	31	-	33	-	9	-	9	-	9	-	9	-	4	-	4
Content Management	1	6	1	6	10	59	3	66	3	53	1	55	2	28	-	30	-	8	-	8	-	4	-	4	-	2	-	2
Social Media Management	1	13	-	14	2	63	-	65	7	31	-	38	2	22	1	23	1	2	-	3	-	3	-	3	-	1	-	1
Browser Automation	-	1	-	1	9	45	-	54	5	29	-	34	2	13	-	15	-	1	-	1	1	7	-	8	-	5	-	5
Design Tools	-	3	-	3	5	59	2	62	-	23	-	23	-	15	-	15	-	5	-	5	-	-	-	-	-	-	-	-
Mobile Development	-	1	-	1	7	51	1	57	1	22	-	23	-	10	-	10	-	3	-	3	-	3	-	3	-	-	-	-
Game Development	-	-	-	-	1	27	-	28	2	35	-	37	-	12	-	12	-	3	-	3	-	3	-	3	-	1	-	1
Learning & Documentation	-	5	-	5	2	36	1	37	2	19	-	21	-	15	-	15	-	1	-	1	-	4	-	4	-	-	-	-
Deployment & DevOps	-	1	-	1	3	25	-	28	-	30	-	30	-	9	-	9	-	1	-	1	-	2	-	2	-	1	-	1
Marketing Automation	-	2	-	2	1	11	-	12	3	12	-	15	2	13	-	15	1	-	-	1	-	2	-	2	-	1	-	1
API Development	10	110	1	119	11	140	-	151	12	127	-	139	2	32	-	34	-	17	-	17	-	13	-	13	-	1	-	1
Official	-	4	-	4	-	1	-	1	-	1	-	1	-	1	-	1	-	-	-	-	-	6	-	6	-	1	-	1
Featured	-	5	1	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Other	14	167	5	176	8	53	5	56	8	61	4	65	1	24	-	25	1	10	-	11	-	2	-	2	-	1	-	1
Unknown	2	23	–	25	3	22	1	24	21	288	2	307	–	26	–	26	20	132	3	149	4	9	–	13	–	–	–	–

(II) Language-Level Analysis of Misuse. We examine how programming languages influence the likelihood of cryptographic misuse. Table 3 summarizes the distribution of programming languages across different MCP markets, along with the presence of cryptographic usage and identified misuses. To avoid skewed interpretations, when we analyze the data, we exclude extreme cases such as Java and Rust, where the absolute number of servers is too small to draw meaningful conclusions. The results highlight two points: (1) Python's misuse density is consistently higher across markets, marking it as a critical focus for remediation; and (2) markets with more Python adoption (e.g., Smithery, Mcpmarket) also report higher overall misuse counts, suggesting a compounding effect between language ecosystem characteristics and market curation practices. For example, in Mcpmarket, Python accounts for 89 crypto-enabled servers with 30 misuses, while JavaScript has a larger base (112) but slightly fewer misuses (17). Interestingly, this trend repeats in Smithery Registry. This suggests that although JavaScript remains the most widely used language, Python implementations are disproportionately prone to misuse, likely due to its extensive but unevenly vetted cryptographic library ecosystem.

(III) Category-Level Analysis of Misuse. We now turn to the analysis of misuse from the perspective of functional categories. As shown in Figure 12, the distribution of misuses is highly uneven, with clear concentration in certain categories. Developer Tools and Data Science & ML stand out as the two dominant categories, together contributing more than half of the observed misuses. This pattern is not surprising, since these categories

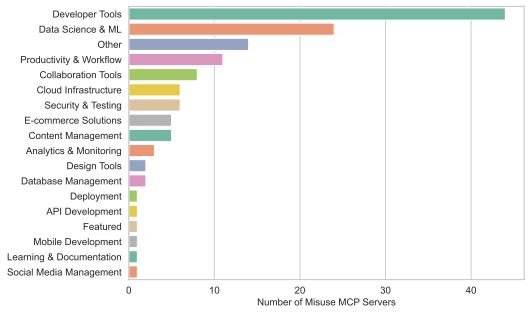
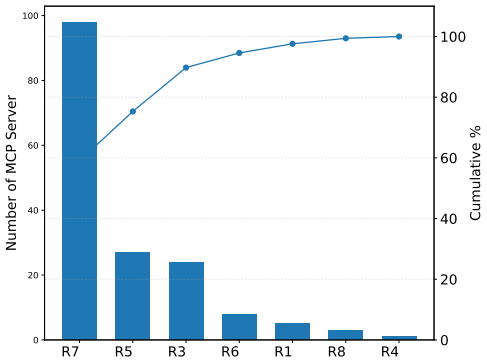
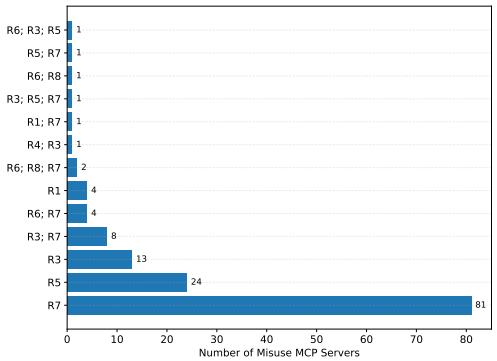


Fig. 12. MCP Servers with Crypto Misuse by Category

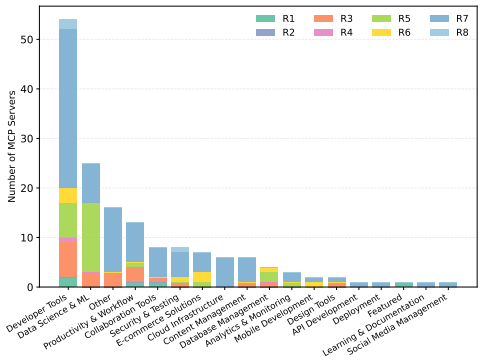
often involve direct handling of sensitive data, code execution, or cryptographic operations, which increases both the need for crypto and the likelihood of insecure implementation. In contrast, categories like Learning & Documentation, Social Media Management, and Mobile Development show only minimal misuses. Tools in these areas are often built upon external APIs or existing frameworks, reducing the chance of developers implementing cryptographic functionality themselves. As shown in Table 4, misuse concentration is both category- and market-dependent. Developer Tools and Data Science & ML consistently emerge as hotspots, while areas like Game Development, Design Tools, and Mobile Development show almost no cases, likely due to limited crypto use or reliance on secure external APIs.



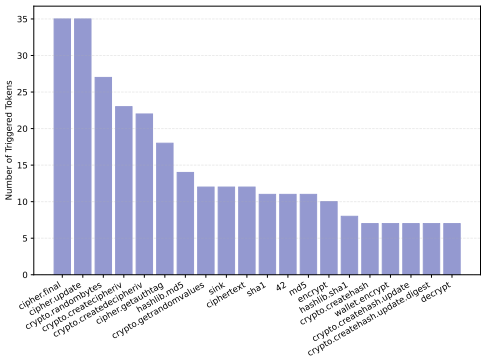
(a) MCP Servers with Crypto Misuse by Rule



(b) Distribution of Multi-Rule Combinations and Single-Rule Violations in MCP Crypto Misuse



(c) MCP Servers Triggering Rules in Each Category



(d) Most Frequent Tokens Triggering Crypto Misuse

Fig. 13. Overview of MCP Rule Misuse Patterns

(IV) Rule-Level Analysis of Misuse. We examine the distribution of misuses across the eight detection rules (R1–R8). As shown in Figure 13a, misuses are concentrated in a few categories, with R7 (Missing Integrity Protection) the most prevalent: servers often apply encryption without authentication (e.g., AES in CBC/CTR mode without MAC/GCM), leaving ciphertext vulnerable to undetected tampering. R3 (Weak Hash Functions) and R5 (Static PRNG Seeds) also stand out,

reflecting continued reliance on MD5/SHA-1 and predictable randomness. Importantly, misuses rarely occur in isolation: Figure 13b shows that the combination of R3 and R7 (weak digests together with missing integrity checks) and the combination of R6 and R7 (insecure block modes with missing authentication) significantly amplify the attack surface. The joint rule–category distribution in Figure 13c further indicates clustering within specific domains. Developer Tools exhibit the broadest misuse spectrum (R1–R7), dominated by R7, R5, and R3, while Security & Testing tools paradoxically also suffer from R7, R5, and R6, often due to static configurations or simplistic client–server protocols. These findings highlight a systemic trade-off of prototyping speed over sound cryptographic design. Finally, token-level analysis (Figure 13d) links these rules to concrete coding practices: `cipher.update/cipher.final` without `cipher.getAuthTag` drive most R7 cases, while `hashlib.md5` and `hashlib.sha1` account for the bulk of weak-hash misuses, confirming that outdated primitives remain pervasive despite long-standing deprecation.

5.4 Case Study and Security Implications

Case Study 1: Leaked LLM API Keys. We identified MCP servers that embed hard-coded API keys for LLM services, manifesting a clear violation of R1 (Fixed Key / API Key). For instance, the *Excel Master MCP Server* includes a plaintext Gemini key, and the dify-for-dsl MCP Server defines a static `API_KEY` (Figure 14). These exposed keys remain active and usable, allowing unauthorized actors to immediately exploit them. Given that Gemini API usage is billed per token processed (for example, the Gemini 2.5 Pro model charges $1.25 per million input tokens and $10 per million output tokens [21]) such misuse can lead to substantial financial losses, even if the unauthorized usage is modest. If a leaked key is used at scale or inserted into automated scripts, the costs could quickly escalate into hundreds or thousands of dollars.

```
HOST = '127.0.0.1'
HOST2 = '192.168.2.2'
PORT = 15012
API_KEY = 'AIzaSyD**********************7n0B7nSgCS9U'
PROXY = 'http://127.0.0.1:7897'
```

```
# Configure Gemini
genai.configure(api_key='AIzaSyAe********************
 ↪  **IBsH1zn4')
model = genai.GenerativeModel('gemini-2.0-flash-001')
 ↪  # Use flash model for structured extraction
```

Fig. 14. MCP Server Configuration (Hardcoded API Key): Excel Master MCP Server (left) vs. dify-for-dsl MCP Server (right)

Case Study 2: Insecure Crypto MCP Server. The *Crypto MCP Server* exposes cryptographic primitives as standardized MCP tools, making them callable by IDEs or AI assistants. However, one of its tools implements DES in ECB mode (Figure 15), a configuration widely considered insecure. DES's small key size makes brute-force feasible, while ECB mode leaks plaintext structure by encrypting identical blocks deterministically. Exposing such a tool through MCP magnifies the danger into a *supply-chain vulnerability*, where insecure primitives can be reused by many downstream applications, severely undermining confidentiality and integrity guarantees.

```
const encrypted = CryptoJS.DES.encrypt(message, keyHex, {
  mode: CryptoJS.mode.ECB,
  padding: CryptoJS.pad.Pkcs7,
});
```

Fig. 15. DES with ECB Mode in Crypto MCP Server

Case Study 3: Weak Hash in 1Panel MCP Server. The *1Panel MCP Server* provides automated deployment capabilities but implements a flawed authentication scheme. Each request header

includes a token computed as `MD5("1panel" + apiKey + timestamp)` (Figure 16). The reliance on **MD5**, a weak hash function, enables efficient offline brute-forcing if a token or API key is leaked. Because MD5 is susceptible to preimage and collision attacks, this design allows adversaries to forge valid tokens and compromise the deployment pipeline. The impact is severe: an attacker could bypass authentication and gain control over the deployment process, directly threatening system integrity.

```
getAuthHeaders() {
  const timestamp = Math.floor(Date.now() / 1000).toString();
  const content = `1panel${this.apiKey}${timestamp}`;
  const token = crypto.createHash("md5").update(content).digest("hex");
  return {
    "1Panel-Token": token,
    "1Panel-Timestamp": timestamp,
    "Accept-Language": this.languageCode,
  };
}
```

Fig. 16. Token Generation Function `getAuthHeaders()` in 1Panel MCP Server

6 DISCUSSION

Limitations. Although MICRYSCOPE demonstrates strong capabilities in systematically uncovering cryptographic misuses across heterogeneous MCP servers, several limitations remain. First, our analysis pipeline is primarily static and relies on intermediate IR construction together with taint propagation. As a result, dynamic aspects of MCP servers (such as IVs generated at runtime from environment randomness or keys derived through user interactions) may not be fully captured, occasionally leading to false negatives. Second, our rule set, while covering eight representative misuse categories, is still bounded by pre-defined patterns. Although such semantic gaps can obscure misuses (e.g., crypto wrapped in custom utility functions), in practice these cases are relatively rare compared to the broader misuse patterns we observed.Third, MICRYSCOPE focuses on code-level misuse and does not yet cover deployment issues (e.g., insecure transport, weak dependencies, outdated libraries), though these can also undermine MCP security but are extendable in future work.

Root Causes of Misuse in the MCP Ecosystem. A central question raised by our study is why cryptographic misuses (19.7%) are so pervasive in the MCP ecosystem. MCP provides only minimal safeguards and lacks authenticity or confidentiality guarantees, pushing developers to implement custom crypto prone to classic pitfalls. Second, the heterogeneity of MCP implementations intensifies this challenge: servers span more than ten programming languages, each with distinct API conventions, implicit defaults, and pitfalls, which complicates the enforcement of uniform secure practices. Third, the plugin-style architecture of MCP tools fosters weak coupling and implicit execution paths. Because LLMs dynamically orchestrate functions at runtime, secure and insecure utilities may coexist, and insecure compositions can emerge only under certain prompt-driven workflows. Finally, the developer-centric and market-driven nature of MCP encourages rapid prototyping and mass publication of servers, where functionality and interoperability often take precedence over cryptographic rigor.

Ethical Considerations. This work focuses on analyzing publicly available MCP server implementations to detect cryptographic misuse. We carefully considered the ethical implications of our methodology and findings, guided by the principles outlined in the Menlo Report and subsequent discussions on ethical frameworks in computer security research.

- **Impacts and Ethical Principles**. Following the principle of Beneficence, we designed our methodology to maximize positive outcomes (identifying and mitigating systemic weaknesses) while minimizing potential harms. Respect for Persons was upheld by avoiding any collection of private user data; our analysis was limited to open-source code and public registries. Justice guided our focus on a broad and representative set of MCP servers, ensuring that security improvements benefit the entire ecosystem rather than a narrow subset. Finally, Respect for Law and Public Interest was maintained by operating only on publicly available artifacts and adhering to repository terms of service.
- **Harms and Mitigations**. We acknowledge potential harms such as reputational damage to developers whose insecure implementations are identified, or the possibility that adversaries could misuse our findings to locate exploitable systems. To mitigate these risks, we (i) did not disclose vulnerabilities in a way that enables direct exploitation, (ii) reported critical cases to responsible parties. For example, we responsibly reported exposed keys to affected vendors. In several cases, vendors promptly revoked or deprecated the leaked credentials following our disclosure. To confirm these actions, we relied on safe, non-inference validation methods such as querying metadata or account-status endpoints [12, 22, 26], which allow us to verify whether a key is active without incurring costs or triggering model inference. This approach ensured that our verification process did not itself introduce risk, while providing evidence that remediation steps were indeed taken.

We conclude that the ethical benefits outweigh potential harms: systematically documenting the prevalence and causes of cryptographic misuse in MCP serves the community by highlighting systemic risks and motivating stronger standards, tools, and practices. The decision to publish is supported both by beneficence (maximizing ecosystem security) and by respect for law and public interest (avoiding unlawful or privacy-invasive actions)

7 RELATED WORK

Cryptographic Misuse. Recent studies show that cryptographic misuse remains a widespread and persistent threat to software security, even in modern development environments [2, 10, 14, 32]. There are various types of misuse, such as the use of weak algorithms (e.g., ECB mode [4], MD5/SHA-1 [30, 34], DES/RC4 [16]) flawed configurations (e.g., fixed IVs [23], fixed seeds [35], or salts [11, 15, 20, 38], and low PBKDF2 iterations [27]). Cryptographic misuse can arise in a variety of domains. For example, Wang et al. [32] found that 95% of 1,431 IoT firmware samples contained misuse. Yu et al. [25] observed that mini-applications are also prone to cryptographic misuse [5, 37]. Many misuses stem from developers' misunderstanding of cryptographic APIs. Therefore, researchers proposed static analysis and rule-based detection as effective solutions. Several studies formalized common misuse patterns to facilitate tool development [4], while others highlighted the need to enhance detection capabilities in the context of LLM-generated code [9, 13, 35]. Our work differs by providing the first systematic study of cryptographic misuse in MCP.

MCP Security. MCP is a standardized interface that connects LLMs with external tools by structuring inputs and coordinating multi-source information [8, 14, 29]. However, recent studies have shown that if an insecure MCP is connected, it can become an exploit path for attackers to control LLM behavior, inject malicious instructions, steal assets, and achieve remote code execution (RCE) [7, 17]. Specifically, untrusted MCP data sources or tool responses can lead to serious risks such as prompt injection [18, 19, 24], cross-service prompt stealing [36], demystifying RCE manipulation of executors [17], and trojanizing plugins [7]. Furthermore, the complexity of responsibility attribution introduced by long contexts requires enhanced traceability in MCP design [33]. Unlike these studies

focusing on prompt injection or malicious plugins, we examine misuse of cryptographic operations inside MCP servers, exposing a complementary layer of risk.

8 CONCLUSION

We introduced MICRYSCOPE, the first framework for detecting cryptographic misuses in MCP implementations. Our large-scale study of 9,403 servers shows that 19.7% of crypto-enabled MCP servers contain misuses, with risks clustering in specific markets, languages, and categories. Common pitfalls create tangible threats from financial abuse to supply-chain vulnerabilities. These findings demonstrate that misuse in MCP is systemic, underscoring the need not only for better detection but also for automated remediation, developer guidance, and protocol-level defenses to ensure MCP can serve as a secure foundation for the agentic AI ecosystem.

REFERENCES

[1] Model context protocol. https://en.wikipedia.org/wiki/Model_Context_Protocol, 2025. Accessed: 2025-08-23.

[2] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 614–631. IEEE, 2022.

[3] Axios. Hot new protocol glues together ai and apps, 2025. Accessed: 2025-08-23.

[4] Yikang Chen, Yibo Liu, Ka Lok Wu, Duc V Le, and Sze Yiu Chau. Towards precise reporting of cryptographic misuses. In *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[5] Yu Chen, Yuanchao Chen, Ruipeng Wang, Taiyan Wang, Shouling Ji, Hong Shan, Dan Xu, and Zulie Pan. Whiskey: Large-scale identification of mobile mini-app session key leakage with llms. *IEEE Transactions on Information Forensics and Security.*, 20:5872–5887, 2025.

[6] Contentful. Model context protocol: The new ai connection standard, 2025. Accessed: 2025-08-23.

[7] Tian Dong, Minhui Xue, Guoxing Chen, Rayne Holland, Yan Meng, Shaofeng Li, Zhen Liu, and Haojin Zhu. The philosopher's stone: Trojaning plugins of large language models. In *Proceedings 2025 Network and Distributed System Security Symposium.*, 2025.

[8] Abul Ehtesham, Aditi Singh, and Saket Kumar. Enhancing clinical decision support and ehr insights through llms and the model context protocol: An open-source mcp-fhir framework. In *2025 IEEE World AI IoT Congress (AIIoT)*, pages 0205–0211, 2025.

[9] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Ryan Tsang, Najmeh Nazari, Han Wang, Houman Homayoun, et al. Large language models for code analysis: Do LLMs really do their job? In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 829–846, 2024.

[10] Konstantin Fischer, Ivana Trummová, Phillip Gajland, Yasemin Acar, Sascha Fahl, and Angela Sasse. The challenges of bringing cryptography from research papers to products: Results from an interview study with experts. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7213–7230, 2024.

[11] Conor Gilsenan, Fuzail Shakir, Noura Alomar, and Serge Egelman. Security and privacy failures in popular 2fa apps. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2079–2096, 2023.

[12] GitHub community. Adding validation for openai api key #655, 2024. Accessed: 2025-08-24.

[13] Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879, 2023.

[14] Sevinj Karimova and Ulviya Dadashova. The model context protocol: A standardization analysis for application integration. *Journal of Computer Science and Digital Technologies*, 1(1):50–59, 2025.

[15] Seungho Kim and Hyoungshick Kim. Privacy-preserving cryptographic api misuse detection framework using homomorphic encryption. *Journal of the Korea Institute of Information Security & Cryptology*, 34(5):865–873, 2024.

[16] Wenqing Li, Shijie Jia, Limin Liu, Fangyu Zheng, Yuan Ma, and Jingqiang Lin. Cryptogo: Automatic detection of go cryptographic api misuses. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 318–331, 2022.

[17] Tong Liu, Zizhuang Deng, Guozhu Meng, Yuekang Li, and Kai Chen. Demystifying rce vulnerabilities in llm-integrated apps. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1716–1730, 2024.

[18] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1831–1847, 2024.

[19] Yupei Liu, Yuqi Jia, Jinyuan Jia, Dawn Song, and Neil Zhenqiang Gong. Datasentinel: A game-theoretic detection of prompt injection attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2190–2208. IEEE, 2025.

[20] Prianka Mandal, Amit Seal Ami, Victor Olaiya, Sayyed Hadi Razmjo, and Adwait Nadkarni. "belt and suspenders" or" just red tape"?: Investigating early artifacts and user perceptions of iot app security certification. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4927–4944, 2024.
[21] Glide AI News. Google gets aggressive on pricing with new gemini 2.5 pro, 2025. Accessed: 2025-08-20.
[22] OpenAI Community Forum. What are the valid characters for the openai api key?, 2024. Accessed: 2025-08-24.
[23] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P Carloni, and Simha Sethumadhavan. Crylogger: Detecting crypto misuses dynamically. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1972–1989. IEEE, 2021.
[24] Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. "do anything now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1671–1685, 2024.
[25] Yizhe Shi, Zhemin Yang, Kangwei Zhong, Guangliang Yang, Yifan Yang, Xiaohan Zhang, and Min Yang. The skeleton keys: A large scale analysis of credential leakage in mini-apps. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.
[26] Stack Overflow community. How to check the validity of the openai key from python?, 2023. Accessed: 2025-08-24.
[27] Cong Sun, Xinpeng Xu, Yafei Wu, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. Cryptoeval: Evaluating the risk of cryptographic misuses in android apps with data-flow analysis. *IET Information Security*, 17(4):582–597, 2023.
[28] SuperAGI. The future of model context protocol: Emerging trends and predictions for mcp server adoption in the next 5 years, 2025. Accessed: 2025-08-23.
[29] Sudarvizhi T, Arjun A Chandran, Gowtham L, and Poomainthan M. Smart air quality monitoring with model context protocol for environmental safety technology. In *2025 3rd International Conference on Inventive Computing and Informatics (ICICI)*, pages 01–07, 2025.
[30] Adriano Torres, Pedro Costa, Luis Amaral, Jonata Pastro, Rodrigo Bonifácio, Marcelo d'Amorim, Owolabi Legunsen, Eric Bodden, and Edna Dias Canedo. Runtime verification of crypto apis: An empirical study. *IEEE Transactions on Software Engineering.*, 49(10):4510–4525, 2023.
[31] The Verge. Windows is getting support for the 'usb-c of ai apps', 2025. Accessed: 2025-08-23.
[32] Jianing Wang, Shanqing Guo, Wenrui Diao, Yue Liu, Haixin Duan, Yichen Liu, and Zhenkai Liang. Cryptody: Cryptographic misuse analysis of iot firmware via data-flow reasoning. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 579–593, 2024.
[33] Yanting Wang, Wei Zou, Runpeng Geng, Jinyuan Jia, and Reviewing Model. Tracllm: A generic framework for attributing long context llms. In *34rd USENIX Security Symposium (USENIX Security 25)*, 2025.
[34] Anna-Katharina Wickert, Lars Baumgärtner, Michael Schlichtig, Krishna Narasimhan, and Mira Mezini. To fix or not to fix: A critical study of crypto-misuses in the wild. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 315–322. IEEE, 2022.
[35] Yifan Xia, Zichen Xie, Peiyu Liu, Kangjie Lu, Yan Liu, Wenhai Wang, and Shouling Ji. Beyond static pattern matching? rethinking automatic cryptographic api misuse detection in the era of llms. *Proceedings of the ACM on Software Engineering.*, 2(ISSTA):113–136, 2025.
[36] Yong Yang, Changjiang Li, Qingming Li, Oubo Ma, Haoyu Wang, Zonghui Wang, Yandong Gao, Wenzhi Chen, Shouling Ji, and Reviewing Model. Prsa: Prompt stealing attacks against real-world prompt services. In *34rd USENIX Security Symposium (USENIX Security 25)*, 2025.
[37] Yue Zhang, Yuqing Yang, and Zhiqiang Lin. Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS, 26-30, 2023*, pages 2411–2425. ACM, 2023.
[38] Yuexi Zhang, Bingyu Li, Jingqiang Lin, Linghui Li, Jiaju Bai, Shijie Jia, and Qianhong 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*, pages 2978–2992, 2024.