Gopher: High-Precision and Deep-Dive Detection of Cryptographic API Misuse in the Go Ecosystem

Yuexi Zhang
zhangyuexi@buaa.edu.cn
School of Cyber Science and Technology, Beihang University
Beijing, China

Bingyu Li*
libingyu@buaa.edu.cn
School of Cyber Science and Technology, Beihang University
Beijing, China

Jingqiang Lin
linjq@ustc.edu.cn
School of Cyber Science and Technology, University of Science and Technology of China
Hefei, China

Linghui Li
lilinghui@bupt.edu.cn
School of Cyberspace Security, Beijing University of Posts and Telecommunications
Beijing, China

Jiaju Bai
baijiaju@buaa.edu.cn
School of Cyber Science and Technology, Beihang University
Beijing, China

Shijie Jia
jiashijie@iie.ac.cn
Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China

Qianhong Wu
qianhong.wu@buaa.edu.cn
School of Cyber Science and Technology, Beihang University
Beijing, China

ABSTRACT

The complexity of cryptographic APIs and developers' expertise gaps often leads to their improper use, seriously threatening information security. Existing cryptographic API misuse detection tools that rely on black/white-list methods require experts to manually establish detection rules. They struggle to dynamically update rules and scale to cover numerous unofficial cryptographic libraries. Furthermore, as these tools are primarily aimed at non-Go languages, they have limited applicability and accuracy in the Go ecosystem, which is extensively used for security-centric applications.

To mitigate these challenges, we present *Gopher*, a novel cryptographic misuse detection framework, that excels in encapsulated API and cross-library detection. In this framework, we have designed *CryDict* to convert rules into unified and standardized constraints, capable of deriving new usage rules and elucidating implicit knowledge during scanning. *Gopher* leverages *CryDict* to create a logical separation between rule formulation and *Detector* detection, enabling dynamic updating of constraints and enhancing detection capabilities. This significantly improves the *Gopher*'s compatibility and scalability. Utilizing *Gopher*, we have conducted an extensive analysis of the Go ecosystem, examining 19,313 Go projects. In our rigorous testing, *Gopher* demonstrated a remarkable 98.9% accuracy rate and identified 64.1% of previously undetected misuses. This scrutiny has surfaced numerous hidden security vulnerabilities, and highlighted misuse tendencies across diverse project categories.

CCS CONCEPTS

• **Security and privacy** → *Malware and its mitigation*; *Software and application security*.

KEYWORDS

Cryptographic API Misuses, Static Analysis, Golang Ecosystem Measurement

ACM Reference Format:
Yuexi Zhang, Bingyu Li, Jingqiang Lin, Linghui Li, Jiaju Bai, Shijie Jia, and Qianhong Wu. 2024. Gopher: High-Precision and Deep-Dive Detection of Cryptographic API Misuse in the Go Ecosystem. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3658644.3690276

1 INTRODUCTION

Cryptography is integral to the realm of information security, serving as the bedrock for safeguarding data confidentiality, integrity, availability, authenticity, and non-repudiation. When cryptographic algorithms are reliably implemented with securely configured parameters, they offer the dependable security guarantee. Especially benefiting from its "*efficient concurrent processing*", "*cross-platform and portability*", and "*rich standard library*", etc., the Go programming language has been extensively utilized for network services and other security-centric applications [11, 30, 57, 58]. The widespread adoption of Go has naturally led to an increased reliance

*Bingyu Li is the corresponding author.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690276

on cryptographic APIs to ensure the security and integrity of data across these services [56].

However, the misuse of cryptographic APIs has become exceedingly common due to a variety of factors [28]. These include the intrinsic complexity of cryptographic APIs [1], a shortage of specialized training in the secure application of cryptography [3], the proliferation of insecure and misleading information on online forums (e.g., Stack Overflow) [41], and an overreliance on potentially risky code examples provided by generative AI (e.g., ChatGPT) [2]. Such misuse significantly undermines the security assurances that cryptographic algorithms are designed to offer [48, 68], seriously threatening information security.

The methods for detecting cryptographic API misuse are mainly bifurcated into static and dynamic analyses. Static analysis tools, including CogniCrypto$_{SAST}$ [36], CrytpoGuard [49], and CryptoGo [39], forgo the need for program execution, rendering them well-suited for swift, extensive detection initiatives. However, they are not exempt from generating a substantial number of false positives. Conversely, dynamic analysis tools like Crylogger [46] offer validation of potential misuses, which contributes to a reduced incidence of false positives. Yet, dynamic tools may inadvertently overlook certain misuses due to code coverage issues [29].

Most of the existing tools rely on a black/white list approach, necessitating manual misuse rule creation by experts, whether for basic APIs or for those encapsulated or developed further. This approach is not only labor-intensive but also limited in scope, as (*a*) it is confined to detecting predefined rules, lacking the capability to support dynamic updating of rules [24, 39, 49], and (*b*) it is difficult to adapt unofficial cryptographic APIs and performing cross-library detection [15]. Although some research has explored the use of AI for misuse detection [17, 65], which often face challenges in autonomously generating reliable and guaranteed rules.

To address these challenges, we introduce *Gopher*, a novel framework for the high-precision and deep-dive detection of cryptographic misuse. Within the framework, we have designed *CryDict*, the cornerstone of *Gopher*, a method that concisely and efficiently translates usage rules into formal constraints; and developed a high-fidelity static *Detector* specifically tailored for the Go. Distinct from traditional analysis paradigms, *Gopher* incorporates a feedback mechanism that empowers the *CryDict* to autonomously generate novel constraints for *non-basic* cryptographic API usage, particularly suitable for encapsulation API and cross-library scenarios. This is achieved by analyzing the data-flow gathered during the *Detector*'s misuse analysis, reducing the overhead of manually designing constraints. This advancement enables *Gopher* to dynamically adapt to updates in detection constrains and improve its detection coverage without secondary development of tools, thereby addressing the scalability challenges faced by tools that rely on static black/white list approaches.

To implement large-scale detection of cryptographic API usage within the Go ecosystem, our work includes the following aspects:

(1) Development of CryDict Constraint Description Specification. To transcend the limitations of existing tools that are restricted to specific detection rules and cryptographic librarys, we have devised *CryDict*, enabling the concise and standardized formulation of rules through three types of *constraints* constructed from a set of ten *predicates*. This create a clear intermediate formalized representation method between the natural language description of cryptographic API usage rules and the specific execution of detection tasks. This intermediate layer shields the upper-level detection tool from the dynamic fluctuations of underlying rules, achieving a decoupling of the tool from the detection rules, *thereby equipping detection tool with dynamic scalability*.

(2) Constraint Derivation and Feedback Integration. A pivotal aspect of our framework is its ability to automatically derive new *CryDict* constraints in real-time during the scanning process using constraint solving methods. This capability not only enables the tool to continuously improve itself during the scanning process, discover more deep-seated misuse, but also enables support for detecting non-official cryptographic APIs and cross-library API invocations. Concurrently, it encapsulates the implicit expertise in cryptographic library utilization into a tangible *CryDict* format, offering developers a clear guide for reference and application, *thereby enabling our detection tool to be universally applicable.*

(3) Enhancements in Other Aspects of the Framework. The framework's improvements extend across several dimensions: (*a*) We have expanded our detection scope to encompass a broader range of complex algorithms and protocols that have been previously underrepresented, such as argon2, bcrypt, and ssh [8, 47, 66]. (*b*) We have bolstered the coverage and efficiency of misuse detection tools by implementing rapid detection algorithms that are sensitive to domains, flows, and contexts. (*c*) We devised a series of optimization methods for reducing false positives, tailored to the specific characteristics of Go and cryptography.

Through the synergistic integration of various key steps, our tool discovered an additional 64.1% of misuses instances within the test dataset, while maintaining a accuracy of 98.9%.

Our contributions are as follows:

- **Framework Innovation**: We introduce a novel framework, *Gopher*, for cryptographic API misuse analysis. This framework can automatically derive potential new constraints for (encapsulated) cryptographic APIs, providing an explicit expression of the inherent implicit knowledge in the use of *non-basic* cryptographic libraries, aimed at dynamically adapting to various cryptographic libraries in misuse detection.
- **Design of CryDict and Development of Detector**: Building upon our framework, we have designed *CryDict*, a formal concise method for describing constraints, supporting both dynamic constrains updates and automatic constrains derivation; additionally, we developed a *Detector* that utilizes *CryDict* for detecting misuse involving non-official APIs and cross-library. The decoupling of *CryDict* from the *Detector* endows it with excellent compatibility, as illustrated by the significant enhancement achieved through its integration with *CryptoGo*.
- **Large-Scale Ecosystem Analysis**: We have conducted an unprecedented large-scale analysis of the Go ecosystem, scrutinizing 19,313 projects, and selecting 7 types of common Go projects, identifying 145 representative cases for in-depth study. Our findings reveal that 67.31% of projects exhibit misuse. We have disclosed the issues we found to the developers and are currently developing a benchmark dataset based on our findings.

```
1  package demo1
2  ...
3  type MyEncrypter struct {
4    KeyLength int
5  }
6  var BasicLength = 1024
7
8  func GenKey(e *MyEncrypter) *rsa.PrivateKey {
9    privateKey, _ := rsa.GenerateKey(
10     rand.Reader, e.KeyLength+BasicLength)
11   return privateKey
12 }
13 ------------------------------------------------
14 package main
15 import "demo1"
16
17 func main() {
18   ...
19   myEncrypter := demo1.MyEncrypter{KeyLength: 128}
20   demo1.GenKey(&myEncrypter)
21   ...
22 }
```

Figure 1: An Encapsulation of rsa.GenerateKey **in demo1.**

2 MOTIVATION AND THREAT MODEL

2.1 Motivation

The encapsulation of basic APIs into novel libraries not only facilitates development but also augments the reusability of code, a practice widely adhered to in the software development community [21]. For example, in the Go programming ecosystem, leveraging external dependencies is commonplace, with 76% of developers incorporating five or more external libraries in their projects[31]. Our subsequent real-world testing, detailed in Section 6.1, revealed that 1,121 (25.9%) instances of cryptographic API invocations across 145 actual projects involved cross-library encapsulation.

Example. As shown in Figure 1, within package demo1, the development team has encapsulated the function rsa.GenerateKey from the standard cryptography library into GenKey (line 8 in Figure 1). This encapsulated function is then referenced in the main package (line 20 in Figure 1), which is part of a different project. As per the 2024 recommendations, it is recommended to set the RSA parameter size to 3072 bits or more [6]. Ignoring the recommendation can result in security breaches, including brute-force attacks and the eventual decryption of encrypted data.

While the encapsulation of cryptographic APIs streamlines the creation of a unified interface and enhances ease of use, it complicates the process of static analysis. Many existing static cryptographic misuse detection tools are designed for languages other than Go and focus on improving the precision and speed of identifying misuse within individual libraries [39, 49, 53]. As shown in Figure 2, they typically follow a linear approach: starting with summarizing usage rules, mapping them to detection mechanisms, and executing the detection process.

However, this approach overlooks the potential to harness feedback from scanning output during the detection phase, which could significantly enhance the tool's capability to detect misuse across various libraries, projects, and cross-library processes. In essence, current tools grapple with two primary challenges in detecting cross-library misuses:

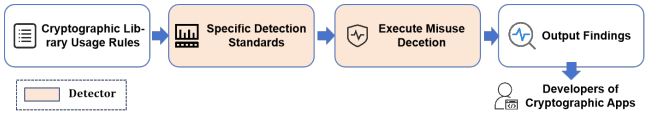


Figure 2: The Framework of Typical Cryptographic API Misuse Detection Tools.

(a) Ambiguous and Complex Cross-Library Data Flow. In large-scale projects, the code-base is often divided into multiple libraries, each compiled independently, making functions from other libraries opaque to the compiler, hindering accurate recognition of behaviors within other libraries (e.g., phantom methods in JAVA) [49]. Even with access to all code, cross-library data flow analysis expands data scope and control flows, escalating analysis complexity. This complexity can lead to a loss of precision and higher demands on time and memory resources for analysis.

(b) Manually Established, Hard-coded Detecting Rules. Most existing tools adopt hard-coded detection rules [24, 39, 49, 53], which are manually crafted and making it challenging to extend the scope of the APIs. Rules based on API names become ineffective when APIs are encapsulated, leading to missed detections [15]. Although *CrySL* in CogniCrypto$_{SAST}$ [36] has partially solved some hardcoding issues, all detection rules still require manual writing by cryptography experts. Given the large number of unofficial cryptographic libraries and complex syntax rules in practice, writing rules for a large scale of unofficial APIs remains practically difficult. Additionally, there is the deliberate obfuscation of function calls by developers aiming to evade detection mechanisms [3].

2.2 Threat Model

Our focus is on the misuse of cryptographic APIs within the Go programming language. It is worth noting that our work focuses on the errors in the invocation of cryptographic libraries rather than errors within the libraries' implementations themselves.

(a) Misuses in calling basic cryptographic libraries. This category focuses on vulnerabilities that arise from the invocation of various *basic* cryptography libraries. In the context of Golang, this involves potential misuses when calling the Go standard library (crypto/...) and Go supplementary library (golang.org/x/crypto/...). These misuses are often related to outdated algorithms and incorrect parameter configurations.

(b) Misuses due to cross-library calls. This type of misuse is more subtle and often goes undetected by current cryptographic API analysis tools. Since problematic code can be located in different library functions, manually setting detection rules for cross-library function calls on a large scale is a complex task.

For the first type of misuse, we conducted a state-of-the-art survey to comprehensively understand the usage rules of basic cryptographic APIs. For the second type of misuse, to circumvent intricate cross-library analysis and manual design of a multitude of constraints, We opted to implement the automatic derivation of encapsulated API constraints at the library level and confine each analysis instance within a single library, reducing the substantial overhead of cross-library data flow analysis by using the derived constraints of the called libraries. Consequently, we propose an innovative framework for cryptographic API analysis.

3 THE FRAMEWORK OF GOPHER

To address these challenges, we present *Gopher*, a novel framework for detecting misuse in cryptographic APIs. As shown in Figure 3, *Gopher* comprises two main components: ***CryDict*** and ***Detector***, which are logically decoupled. *CryDict* generates and derives constraints; while *Detector* performs misuse detection based on these constraints, and provides the necessary data flow information for *CryDict* to further derive and evolve new constrains. This integration creates a feedback mechanism where constraints are updated and reapplied in a closed-loop system. It should be noted that within *Gopher*, *constraints* represent a formalized and standardized depiction of the *rules* governing the use of cryptographic APIs; furthermore, constrains are constructed through *predicates* (see Section 4.3 for details).

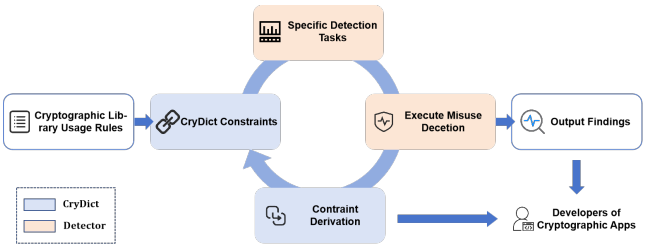


Figure 3: The Framework of Gopher.

(a) Summarizing Basic Cryptographic Library Usage Rules. Gopher initiates with a comprehensive summary of *basic* cryptographic API usage rules, including prevalent implementation patterns and industry standards. Although this initial process requires manual effort, similar to existing tools, the robust nature of *basic* cryptographic algorithms — illustrated by the transition from DES to AES over decades [40], ensures that maintaining and updating these *basic* rules is not the significant burden for *Gopher*.

(b) Building CryDict Constraints. CryDict manually initializes constraints of *basic* API based on summarized rules, and auto-derives *encapsulated* API constraints. These constraints serve as an intermediate, formalized representation bridging the gap between natural language descriptions and specific detection tasks, forming a dynamic constraint library. The separation of constraints from tool enables the adaptation to changes in cryptographic libraries without the need for tool modifications.

(c) Mapping Constraints to Specific Detection Tasks. The aim of this step is to establish a mapping strategy that links *CryDict* constraints with targeted detection tasks, such as taint analysis or program slicing, and to craft specialized analytical methods for different types of constraints. This process is an one-time design effort during the tool's development stage. Afterward, updating *CryDict* constraint is all required to deal changes in the scope of API detection, eliminating the necessity for any further tool modifications.

(d) Executing Misuse Detection. This step executes misuse detection tasks and to collect data flow for subsequent constraint derivation. When conducting cross-library detecting, it is only necessary to load the target (encapsulated) API constraints of the called library. However, it is designed in a completely decoupled manner, meaning that any static or dynamic detection techniques and data flow analysis technologies capable of recognizing *CryDict* constraints can be adapted, such as slicing and taint analysis techniques, ensuring *Gopher*'s compatibility and extensibility.

(e) Constrain Derivation. Based on the data flow information generated during the misuse detection process, we symbolically represent function parameters. By resolving the outcomes of symbolic execution, we deduce novel *CryDict* constraints for the utilization of diverse APIs. This automated derivation significantly diminishes manual effort required for constraint authoring.

The integration of a feedback control mechanism into our framework yields two primary advantages:

Developer Guidance. Studies on cryptographic library availability have often found that the software and its documentation can be opaque, posing a challenge for non-specialists [1, 10, 35, 43]. By analyzing scanned code, *Gopher* can derive new usage constrains, making the implicit usage norms of cryptographic libraries explicit. This provides developers with clear references, enhancing the security of cryptographic applications.

Dynamic Adaptability. Equipped with a responsive feedback mechanism, our tool learns from its detection experiences. It dynamically refines the *CryDict* constraints using real-world data, enabling it to adapt to emerging cryptographic library usage patterns. This process not only minimizes the oversight of potential misuses but also drives the ongoing refinement and optimization of the tool's capabilities.

4 CRYDICT: CONCISE AND EFFICIENT CONSTRAINT SPECIFICATION

In this section, we first categorize common scenarios of cryptographic API misuse and their potential security threats within the Go ecosystem. To surpass the constraints of existing detection methodologies, which are often limited to specific, hard-coded rules, we introduce *CryDict* — a formalized and efficient method for describing cryptographic usage constraints.

4.1 Misuse Scenarios and Cryptographic Rules

Studies analysis of cryptographic library usage reveals that misuse often stems from a lack of comprehensive understanding of API implementations among developers [3, 27, 61]. To enhance the detection of cryptographic misuse, we conducted a thorough examination of scenarios within the Go ecosystem as shown in Table 1. These rules are informed by a combination of scholarly research and authoritative guidelines, such as NIST [6]. We have also utilized existing methods [49, 62] to discuss the severity level for each misuse scenario, see Appendix A.1 for details.

4.2 CryDict Design Principles

To ensure *Gopher* transcends the limitations of only identifying official cryptographic libraries, we devised a unique method for defining proper cryptographic library usage. The complexity of existing constraint languages, which can be challenging to translate into simplified formal rules for correct API usage, led us to introduce *CryDict*. *CryDict* is a specialized constraint description method designed to formally specify the appropriate use of cryptographic APIs. For improved accessibility and ease of understanding, *CryDict*

Table 1: Misuse Scenario of the Go Cryptographic Library.

ID	Misuse Scenario	Severity	Predicate
R01	Low salt length in password hash	L	BYTE_LENGTH
R02	Too short key (e.g., 64-bit key in HMAC)	L	BYTE_LENGTH
R03	Parameter with too small scale (e.g., RSA-512)	L	GEQ
R04	Low iteration parameter in password hash	L	GEQ
R05	Dangerous algorithm (e.g., DES)	H	Function Detector
R06	Warning algorithm (e.g., 3DES)	L	Function Detector SECURE_HMAC_HASH
R07	Use predictable IO to generate parameters	M	RANDOM_IO
R08	Predictable/constant salt in password hash	L	RANDOM_BYTES
R09	Key is predictable/constant/used	H	RANDOM_BYTES
R10	IV in CBC/CFB is predictable/constant	M	RANDOM_BYTES
R11	IV in OFB/CTR/GCM is constant	M	NOT_CONST
R12	Use HTTP protocol	H	HTTPS
R13	Skip the certificate verification in TLS	H	EQ_FALSE
R14	Not verify the client in SSH	H	EQ_FALSE
R15	Using deprecated TLS suites (e.g., TLS_RSA_WITH_AES_128_CBC_SHA256)	H	SECURE_TLS_SUITE
R16	Outdated signature algorithm in TLS (e.g., PKCS1WithSHA1)	H	SECURE_TLS_SUITE
R17	Outdated TLS version (< tls1.2)	H	SECURE_TLS_SUITE GEQ
R18	Insecure suite in SSH (e.g., diffie-hellman-group1-sha1)	H	SECURE_SSH_SUITE
R19	Deprecated functions in go std library	L	Function Detector

utilizes the JSON format. The development of *CryDict* was guided by several key principles:

(1) Blacklist: *CryDict* focuses on identifying a subset of algorithms/functions known to be insecurity, allowing all other functions to be considered security by implication. This approach streamlines document and enhances usability.

(2) Parameter Independence: the requirements for different parameters (or fields) within a function (or object) are often independent, *CryDict* treats each parameter's constraints separately, simplifying the complexity of the rules. For example, in the function rsa.GenerateKey(*random* io.Reader, *bits* int), the requirement for *random* is to utilize cryptographic secure IO, while *bits* is advised to exceed 3072 in 2024. They are independent of each other.

(3) Concurrent Condition Satisfaction: in cryptography, a function call is deemed secure only if all its parameters satisfy all security guidelines. *CryDict* is designed under the premise that all constraints are in a logical "and" relationship, ensuring correct function utilization only when each parameter fulfills its security prerequisites.

(4) Cryptography-related and Tool-independent: *CryDict* is specifically tailored for cryptographic applications. It uses a predefined set of predicates to express constraints, which are crafted to be relevant to cryptographic usage, and the semantics of *CryDict* are detached from the specific analysis methods. This design allows *CryDict* to be utilized with various detection tools, including static/dynamic analysis tools.

4.3 Predicates used in CryDict

CryDict is equipped with a set of 10 built-in *predicates* that provide the necessary expressiveness to define constraints on cryptographic library usage, while maintaining ease of use for developers. These predicates are designed to be simple and understandable. They include eight unary predicates, which take a single argument (either a parameter or a structure field), and two binary predicates (e.g., GEQ), which compare two values. Table 8 in Appendix A.2 presents the predicates used in *CryDict*. Table 1 list the misuse rules they address.

These predicates form the constraints within *CryDict*, allowing for a structured approach to defining rules that ensure the secure application of APIs. The design of *CryDict* predicates aims to balance expressiveness with simplicity, enabling developers to create and understand constraints with minimal complexity. *Predicates* in *CryDict* are declarative statements about subjects, and *constraints* represent the application of these predicates to specific objects or variables.

4.4 CryDict Constraint Specification

To streamline the creation of rules, *CryDict* constraints are defined at the package level, aligning with Go's official documentation conventions. Specifically, a single *CryDict* document outlines the usage rules for an cryptographic package, beginning with the declaration of the package name. Combining the rules and characteristics of the Go cryptography library, *CryDict* has defined a total of three types of constraint specifications, each tailored to address various aspects of cryptographic API usage.

Function_Constraints. This category is primarily dedicated to imposing constraints at the function level. Based on the vulnerability of the function, it can be further divided into two subcategories: (*a*) *DangerousFun_Constraints*, this category includes functions that are deemed to be insecurity for use in cryptographic operations. This includes all dangerous algorithms (e.g., MD5, DES) as shown in Figure 4, lines 3-7. This categorization is grounded in the blacklist principle; (*b*) *WarningFun_Constraints*, they are functions that, while not strictly prohibited, are encouraged in favor of more secure alternatives. This includes algorithms like 3DES, as well as functions deprecated by Go standard library.

Parameter_Constraints. Specifies the conditions under which a function (or method) can be securely used by defining constraints on its parameters. As shown in Figure 4, lines 11-18 mandate that the *bits* parameter in rsa.GenerateKey(*random* io.Reader, *bits* int) must satisfy the predicate GEQ with the value 3072, ensuring the parameter is no less than 3072.

Field_Constraints. Outlines the constraints on the fields of a structure. As shown in Figure 5, lines 7-9 specify that Config.CipherSuites must comply with the predicate SECURE_TLS_SUITE, signifying the use of secure TLS suites.

4.5 Deriving New CryDict Constraints

Traditional approaches to misuse detection have relied heavily on experts manually crafting detection rules, a process constrained by the proliferation of third-party libraries and an inability to scale across various projects and libraries. To transcend these limitations,

```
{
  "Package_Name": "crypto/rsa",
  "DangerousFun_Constraints": [
    {
      "Name": "EncryptPKCS1v15",
      "Reason": "..."
    }
  ],
  "WarningFun_Constraints": "...",
  "Parameter_Constrains": {
    "GenerateKey": [
      {
        "Predicate": "GEQ",
        "Object": ["1", "3072"],
        "Reason": "..."
      },
      "..."
    ],
    "other func": "..."
  }
}
```

Figure 4: CryDict Constraints of crypto/rsa**.**

```
{
  "Package_Name": "crypto/tls",
  "...": "...",
  "Field_Constrains": {
    "Config": [
      {
        "Predicate": "SECURE_TLS_SUITE",
        "Object": ["CipherSuites"],
        "Reason": "..."
      },
      "other fields"
    ]
  }
}
```

Figure 5: CryDict Constraints of crypto/tls**.**

we have explored the automation of cryptographic library constrain derivation based on data-flow information. To avoid generating excessive unhelpful alerts for repairs, we did not apply the constraint derivation to *Function_Constraint*. We provide a detailed explanation in 6.2.

As mention in Section 3, in order to maintain logical decoupling and scalability, the generation and application of *CryDict* constraints are relatively independent, with the latter depending on the implementation of the *Detector* (see Section 5 for details). In this section, we focus on the *CryDict* constraints derivation.

Notation. We employ the following notation to explain the constraint derivation process for the function GenKey, as depicted in Figure 6. We denote nodes in the data propagation process as $n \in N$, which typically correspond to a variable. Our data flow analysis is domain-, flow-, and context-sensitive; hence, our nodes incorporate context and domain information: $N = \text{ssaValue} \times \text{Context} \times \text{Field}$.

For nodes n, $\overrightarrow{n_{in}}$ donates the set of the inputs nodes of node n, with n_{in}^k being the k-th input, and $\overrightarrow{n_{in}} \subseteq N^*$. Each node n may correspond to multiple input nodes. For example, for an addition operation represented by "n_+" (t3 in Figure 6), there are two input nodes, whereas for a constant node (e.g., 4096 in Figure 6), there is no input node. $Res(n)$ represents the set of possible values corresponding to node n. $Res(n) = f_n(\vec{x})$, where $\vec{x} = (x^1, x^2, ..., x^n)$, $x^k = Res(n_{in}^k)$, f_n is determined by the operations performed by the node (e.g., for the "+" operation, $f_{n_+}(\vec{x}) = \{r | r = r_1 + r_2, r_1 \in x^1, r_2 \in x^2\}$).

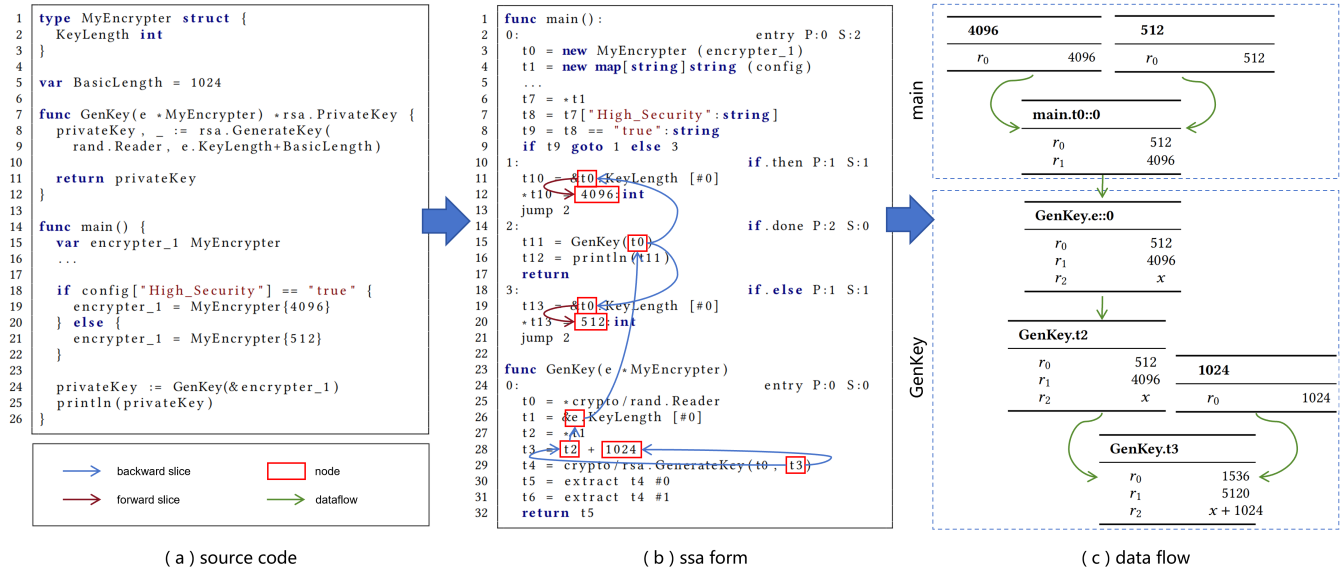
(1) Symbolic Execution in Constrain Derivation. Symbolic execution, a well-established technique in program analysis, involves tracking the influence of inputs on the execution path [16]. Our adaptation of symbolic execution, however, diverges from the standard practice by focusing on function parameters as symbolic entities, disregarding of path conditions, during slicing and data-flow information propagation. This tailored approach facilitates faster examination of the cryptographic facets of the code.

(2) Deriving new CryDict Constraints. Given a parameter Pa that involves existing constraints ϕ, we now explain how to generate new constraints C such that satisfies ϕ. To achieve this, we symbolically represent the parameter node and trace the symbolic execution along the pathway of data dissemination. We denote the outcome set of symbolic execution for any node n as $O(n)$, which serves as the basis for formulating new constraints for the cryptographic library. The new constraints represented as: $C(Pa) := \bigwedge_{r \in O(n_{\text{slice\_criteria}})} (P(r) = \text{True})$.

As an illustration, suppose the existing constraints include $\phi = \bigwedge_{r \subseteq Res(n_{t3})} \texttt{GEQ}(r, 3072)$. Through the analysis results of the detection stage, we can know that the parameter e.KeyLength of GenKey (line 26 in Figure 6(b)) could potentially affect the value of t3. We first abstract the parameter e.KeyLength as a symbolic variable x, at this time $Res(n_{\text{e.KeyLength}}) = \{x, 512, 4096\}$. As data flow propagation, we update the values of other nodes along the propagation path, augmenting the existing results with symbolic execution outcomes. We observe $Res(n_{1024}) = 1024$ and $Res(n_{t3}) = \{1024 + x, 1536, 5120\}$, leading to $O(n_{t3}) = \{1024 + x\}$. Combining with predicate information, we can deduce further constraints: $C(\text{e.KeyLength}) := 1024 + x \geq 3072$. Utilizing SMT solvers, we determines the minimum value of x is 2048. The new *CryDict* constraints derived by *Gopher* as shown in Figure 7. The notation Objects : ["0 :: 0", "2048"] signifies that the 0-th field of the 0-th parameter (indexed from 0) in the GenKey must have a value greater than or equal to 2048.

(3) Leveraging constrain derivation for Cross-library Analysis. Upon completing the analysis of each encountered cryptographic library, *Gopher* archive all the resultant *CryDict* constraints and use them as slice criteria for the slicing process when analyzing other libraries. It should be emphasized that *Gopher*'s constraint derivation for these encapsulated APIs covers all usage information of these APIs, just as a cryptography expert manually specify rules for all new cryptographic libraries. Therefore, when performing cross-library detection, *Gopher* can focus solely on intra-library analysis tasks. For called cross-library APIs, it only needs to verify whether the encapsulated API is invoked as described by the derived constraints, without the need to reload the IR and re-analyze the data flow information of the cryptographic library. This approach streamlines the analysis process and enhances efficiency.

For example, when we identify invocations to demo1.GenKey in other packages and even within different projects, then evaluate if each call to demo1.GenKey meets the specified condition $\bigwedge_{r \in Res(n_{e::0})} \texttt{GEQ}(r, 2048)$, thus facilitating comprehensive cross-library analysis.



(a) source code (b) ssa form (c) data flow

Figure 6: An Example of Detecting the Parameter Scale of RSA Algorithm.

```
{
  "Package_Name": "demoproject/demopkg",
  "Parameter_Constrains": {
    "GenKey": [
      {
        "Predicate": "GEQ",
        "Object": ["0::0", "2048"],
        "Reason": "...",
        "InferDepth": 1
      },
    "other constrains":"...",
}
```

Figure 7: New CryDict Constraints Derived by Gopher.

4.6 Trade-offs in the Design of CryDict

CryDict has deliberately streamlined its expressive capabilities to maintain the clarity of the constrains, yet this has inevitably led to *Gopher* encountering challenges in small number of scenarios.

For example, to simplify the constraint set, *CryDict* was designed adherence to the principle of "independence", implying that the requirements for different parameters could be treated independently (see Section 4.2 for details). This approach, however, presented difficulties for functions similar to scrypt that possess multiple configurable parameters: the minimum memory cost factor (N), the block size (r), and the level of parallelism (p). Such functions boast a multitude of secure configurations. e.g., $N = 2^{17}$, $r = 8$, $p = 1$ and $N = 2^{16}$, $r = 8$, $p = 2$ are both viable secure configurations. This outcome stems from cryptographic libraries typically offsetting decreases in one parameter with enhancements in another.

Recognizing the modern computers' greater tolerance for memory costs over factors like thread count, time, and block size, we tailored *CryDict* with default parameter sizes documented in Golang, with a notable exception for the memory cost. For example, while the official document prescribes the default parameters as $N = 2^{15}$, $r = 8$, $p = 1$ (standard established in 2017). We referred to OWASP's recommendation [44] and tailored our detection standard to $N = 2^{17}$, $r = 8$, $p = 1$ to reflect contemporary advancements in computational capacity. This simplification may result in a minor increase in false positives in scenarios similar to the one below ($N = 2^{16}$, $r = 16$, $p = 2$), but will not introduce false negatives.

```
tempPasswd, _ = scrypt.Key([]byte(passwd), []byte(salt),
        65536, 16, 2, 50)
```

This trade-off in simplifying the expressive has proven to be effective, as evidenced by the results of our large-scale experiments (see Section 7.1 for details). This balance facilitates easier comprehension and application by a broader range of developers, not just cryptography experts, e.g., in reading or authoring new constrains. In addition, to handle the loop paths during symbol execution, we limited the maximum number of data propagation times for each node (see Appendix A.3).

5 INSTANTIATING GOPHER: DETECTION OF CRYPTOGRAPHIC MISUSE

In this section, we instantiate *Gopher*, integrating static analysis techniques to implement the *Detector* within *Gopher* for identifying misuse of cryptographic libraries as defined by constraints within *CryDict*. Initially, we align the constraints specified in *CryDict* with specific analysis tasks for Go programming. Following this, we introduce a method for analyzing data streams. The data flow information gathered during this process will serve as the basis for deriving new *CryDict* constrains. Finally, we demonstrate the extension and optimization of the *Detector* during the instantiation of *Gopher*.

5.1 Mapping Constraints into Analysis Tasks

The *Detector*'s role is to monitor cryptographic API calls against the defined *CryDict* constraints, identifying potential misuses within the codebase. The map of *CryDict* constraints into analysis tasks is a critical step in our method.

(1) Detection Methods for Three Types Constraints. For the enforcement of rules within the *Function_Constraints* category, *Detector* scrutinizes every cryptographic API call within the program. When an API call matches an entry on the pre-established blacklist, *Detector* promptly flags it as a cryptographic misuse, thereby providing a clear indication of potential security vulnerabilities.

We use *program slicing* methods to detect *Parameter_Constrains* and *Field_Constrains*. *Detector* utilizes the arguments associated with function calls as the criteria for backward slicing for *Parameter_Constrains*. "slicing criterion" dictates the commencement of the slicing procedure [64]. Following this, a data flow analysis is executed to assess whether the runtime values associated with these slicing criteria contravene the specified constrains. Any violation is marked as cryptographic misuse by *Detector*. As shown in Figure 6(b), where t3 in line 29 serves as the slicing criterion.

A similar strategy is applied to the constraints defined within *Field_Constrains*. In this context, we employs variables within assignment statements as the slicing criteria for objects of the respective type. This targeted approach is instrumental in determining whether the potential runtime values of these variables conflict with the constrains.

The slicing criterion is formally denoted as $\langle \text{instr}, v \rangle$, where instr represents either a function call or a storage instruction for a variable of a specific type, and v corresponds to the associated variable. After conducting a data flow analysis, if a variable v is found to correspond to multiple runtime values $r \in R$, the *Detector* deems the cryptographic library invocation secure only if all resulting values satisfy the constrains, i.e., $\bigwedge_{r \in R} P(r) = \text{True}$, P is the predicate in *CryDict*. This ensures a stringent condition for the secure use of cryptographic functions. For binary predicates (e.g., `GEQ`), where the second argument's value (e.g., 3072) is provided by *CryDict* constrains, *Detector* only need to track the value of the first argument.

(2) Different Characteristics Detected for Different Predicates. For `RANDOM_BYTES`, *Detector* assesses the randomness of byte slices, verifying their origin from a cryptographically secure source, such as `rand.Read()`, key derivation function (KDF), or any function that fulfills the `RANDOM_IO` predicate (e.g., `io.ReadFull(`r `Reader,` buf `[]byte)`). The crypto/rand package in Go provides a cryptographically secure random number generator. Additionally, *Detector* scrutinizes for any duplicate usage of random values, such as the identical byte slice being repurposed for both the key IV in symmetric encryption.

The *Detector* also reviews the employment of TLS and SSH suites through the `SECURE_TLS_SUITE` and `SECURE_SSH_SUITE`, safeguarding against the use of insecure cryptographic practices. The other predicates are relatively straightforward and easy understand.

5.2 Constructing Data Flow Based on Requirement

To efficiently trace data flow information within a program, we follow a structured method.

(1) Conversion to Intermediate Representation (IR). We employ a partial Static Single Assignment (SSA) approach [52] to translate the source code into an IR format, which ensures each variable is both defined before use and assigned precisely once, thereby streamlining the data flow analysis process. For example, Figure 6(b) displays the SSA form of Figure 6(a).

(2) Execution of Slicing. Our goal is to identify elements that may impact the slicing criterion. Leveraging SSA to establish def-use relationships, *Detector* can easily perform backward slicing. The main difficulty comes from finding possible aliases for variables. We perform local forward slicing on various instruction types, including: (*a*) `Store`, which signify pointer assignments, (*b*) `Unop`, extracting pointer values, (*c*) `ChangeInterface`, utilized in interface creation, (*d*) `Slice`, for generating slices, (*e*) `FieldAddr` and `IndexAddr`, addressing structure fields and slice values, and (*f*) `Call`, which allow for the exploration of orthogonal functions.

This examination aids in uncovering aliases and potential assignments during the backward slicing process. For example, in Figure 6 (b), through forward exploring main.t0, *Detector* find two Store instructions on lines 12 and 20 affecting the data flow.

(3) Inter-procedural Slicing. We use Class Hierarchy Analysis (CHA) to delineate function call relationships, vital for tracking function parameters and identifying call sites. In Go, unlike Java, functions are dynamically resolved only when invoked through an interface. For typical method calls, like "object.method()", the target function can be ascertained at compile time. CHA maintains accuracy while maintaining high efficiency. Once function parameters are traced, *Detector* pursue potential call points by CHA, employing a field- and flow-sensitive slicing approach, with context-sensitivity in the case of orthogonal function exploration.

(4) Sparse Data-flow Information Propagation. Considering the sporadic usage of cryptographic libraries, we initially employed program slicing techniques to pinpoint a minimal set of critical propagation nodes affecting pertinent variables, drastically reducing irrelevant code. Sparse nodes offer two primary advantages. Firstly, they curtail the propagation of invalid data during misuse detection, which significantly bolsters the precision and efficiency of the detection process. Secondly, they mitigate the times of symbolic execution in the subsequent phase of constrain derivation, thereby facilitating a more streamlined and resource-efficient analysis.

Slice results clarify relationship between nodes' inputs and outputs. We start from the node with $|\overrightarrow{n_{in}}| = 0$ (e.g., constant node) and update outcomes of other nodes in sequence as the data propagates. After the propagation is completed, *Detector* determine each possible value of the slice criterion node. We provide constraint conditions for the correct invocation of APIs: $\phi(P, n_{\text{slice\_criteria}}) := \bigwedge_{r \subseteq Res(n_{\text{slice\_criteria}})} (P(r) = \text{True})$.

If the conditions are not met, the *Detector* will flag the instance as a misuse. As depicted in Figure 6(c), setting `encrypter_1.KeyLength` to 512 on line 21 result the runtime value of t3 possible equal to 1536, which violates the established constraint (`GEQ(`r`, 3072)`), triggering a misuse report.

5.3 Updating Cryptographic API Usage Rules

As the field of computer science and cryptography advances rapidly, maintaining the currency of cryptographic algorithm parameters is essential. Our approach to optimization involves three key areas: updating *algorithms*, *parameters*, and *protocols*. This section delves into the nuances of parameter updates, with additional insights on algorithm and protocol updates available in Appendix B.

Neglected Parameters in Historical Analysis. Historically, the focus has been on the parameter settings of public key and hash algorithms, with less emphasis on other frequently utilized cryptographic algorithms, such as password hash and MAC. To address this oversight, we have reviewed scholarly literature and guidelines from organizations like NIST and OWASP. This comprehensive review has culminated in a compilation of recommended parameters for prevalent cryptographic algorithms that are aligned with current computational capabilities (see Appendix B.2 for details).

Updating Outdated Cryptographic Standards: The Case of PBKDF2. PBKDF2 is a key derivation function endorsed by NIST, enjoys widespread adoption across diverse programming languages. Existing detection tools typically set the standard of iteration count for PBKDF2 between 1,000 and 2,000. Initially, the advised minimum iteration count was 1,000 [32]. However, with the evolution of computational capabilities, the OWASP guidelines in 2023 have since elevated the recommended iteration count to 600,000 for PBKDF2-HMAC-SHA256 and to 210,000 for PBKDF2-HMAC-SHA512 [44]. This update renders previous standards obsolete, prompting us to align our detection standards with these more secure thresholds.

5.4 Refining Precision: Mitigate False Positives

Achieving high precision in cryptographic misuse detection is paramount, and our *Gopher* employs several innovative strategies to reduce false positives. This section highlights one key method that leverages the unique characteristics of the Go language and cryptographic algorithms. For a comprehensive overview of additional methods, please refer to Appendix C.

OP1: Leveraging Go's Exception Handling Mechanism. Unlike languages like Java or Python, which utilize structured exception handling with `try/catch` and `try/except` constructs, Go employs a distinct error management strategy. In Go, functions that could generate errors typically return an `err` value, deviating from the exception-throwing paradigm common to many other languages. This feature is pivotal in our strategy to minimize false positives.

Our exploration of this topic was catalyzed by an analysis of code from the `Grafana` [14]. As shown in Figure 8, the `entryKeyToBytes` function mandates that the `salt` value meet the `BYTE_LENGTH(salt, 128)` criterion. Yet, a path-insensitive slicing technique might incorrectly flag a zero-length string returned by `GetRandomString` at line 5, despite Go's exception-handling mechanism effectively preventing execution with an empty salt value, as demonstrated in the `Encrypt` function from lines 15 to 17.

To address this, we have integrated a false positive filtering condition that leverages Go's exception-handling mechanism. During data flow analysis and function tracing, if an output is identified as originating from a null value (e.g., `nil`, " ", or `[]byte` with length 0), The *Detector* evaluates whether the path generated by the slice includes the handling statement for abnormal results (e.g., lines 15-17 in Figure 8). The presence of such handling prompts *Gopher* to disregard any raised misuse alerts.

6 EVALUATION AND SECURITY INSIGHTS

6.1 Selecting the Test Collection

In our preliminary phase, we amassed a collection of 90,326 Go projects that integrate cryptographic libraries, sourced from https://pkg.go.dev. We then meticulously curated this collection, narrowing it down to a refined set of 19,313 projects based on the following stringent criteria to ensure viability.

(1) *Recency and Activity*: Projects must have been updated on GitHub beyond the year 2023, indicating either recent initiation or ongoing active development by its contributors.
(2) *Dependency Management*: The presence of a go.mod file within the project, attesting to the utilization of Go Modules for dependency management.
(3) *Compilability*: The project must compile without errors, a fundamental requirement signifying its practicality for analysis.

Applying our criteria excluded outdated or immature projects, sharpening our analysis' accuracy and efficiency. Notably, we omitted Go test code, which often hardcodes keys for testing, as it doesn't indicate real security issues.

```
1  func GetRandomString(n int, alphabets ...byte)
       (string, error) {
2    ...
3    for i := 0; i < n; {
4      if _, err := rand.Read(randbytes); err != nil {
5        return "", err
6      }
7      ...
8    }
9    return string(bytes), nil
10 }
11
12 func Encrypt(payload []byte, secret string)
       ([]byte, error) {
13   salt, err := GetRandomString(saltLength)
14   // perform exception handling here
15   if err != nil {
16     return nil, err
17   }
18   key, err := encryptionKeyToBytes(secret, salt)
19   ...
20   return ciphertext, nil
21 }
```

Figure 8: Exception Handling in Grafana.

6.2 Comparative Analysis with Existing Tools

Due to the laborious and time-consuming nature of manually excluding false positives in static analysis, conducting an exhaustive in-depth analysis and validation comparison with all 19,313 projects using our tool was impractical. Therefore, we opted for a small-scale targeted experiment utilizing a selection of representative, widely-used and ordinary Go projects within the community. This targeted method enabled a comprehensive and authentic assessment of *Gopher*'s cryptographic API misuse detection capabilities, specifically in comparison with the *CryptoGo* [39], the most advanced publicly available tool dedicated to detection in Go.

Small-scale Targeted Experiment. Our comparative analysis, structured in three tiers, targeted Go projects varying by popularity and usage: (*a*) *highly-stared* (with over 20,000 GitHub stars), (*b*) *highly-imported* (imported by over 750 projects), and (*c*) *ordinary* projects (randomly selected). This stratified sampling ensured a balanced assessment, encompassing projects that are both widely recognized and heavily utilized projects, featuring sophisticated applications of cryptographic APIs, as well as common projects reflecting typical Go ecosystem usage patterns. The distribution

```
1  package demo1
2  func func1() {          // encapsulate rc4 here
3    rc4.NewCipher(key)    // alert
4  }
5  ------------------------------------------------
6  package demo2
7  import "demo1"
8  func func2() {
9    demo1.func1()         // not alert
10 }
```

Figure 9: An Example of Meaningless Constraint Derivation.

of the Go projects is detailed in Appendix D.1. Notably, less than 10% of the projects amass over 90% of the total imports and GitHub stars, underscoring the significance of a detailed examination of widely utilized projects.

We selected 50 projects for each of the three categories; for *highly-starred* and *highly-imported*, we chose the top 50 ranked projects, and for *ordinary*, we randomly selected 50 projects. After eliminating duplicate entries, our final dataset comprised 145 unique and representative test projects for this study.

Within the 145 projects, a total of 4,850 locations were identified for cryptographic API invocations. Our analysis of these API invocations included a review of 525 instances that involved potentially dangerous (warning) function calls, as well as an examination of 4,325 instances related to the verification of parameter or field values. It is noteworthy that within this latter group, 1,121 (25.9%) instances specifically required cross-library parameter validation.

Our primary goal in designing the constraint derivation mechanism is to identify API parameter configuration errors involved cross-library encapsulation, rather than focusing on erroneous invocations of known dangerous or warning functions, such as those specified in R05, R06, and R19. Consequently, we have tailored the scope of *Constrain Derivation* to exclude these specific rules. As illustrated in Figure 9, we opted not to trigger warning when func1() called in func2(). Such warnings tend to be redundant, as the security risk they highlight is generally addressed by focusing on the call to rc4 within func1(), rather than the invocation to func1(). Unnecessary warnings can diminish the effectiveness of developers' remediation efforts.

6.2.1 Accuracy Comparison. To maintain equitable conditions in our evaluation, we documented the fundamental test outcomes for 13 shared rules (3, 5-13, 15-17) that are encompassed by *Gopher* and *CryptoGo*. Although both tools cover R19, which pertains to deprecated functions, it was evaluated separately due to the considerable time elapsed since *CryptoGo*'s last update regarding deprecated functionalities. A detailed examination of the comparative results is presented in Table 2.

Gopher demonstrates a marked superiority across the three categories of tests. Compared to *CryptoGo*, *Gopher* not only sustains a lower rate of false positives (only 11 occurrences, amounting to 1.10%) but also uncovers 385 (an increase from 601 to 986, +64.1%) new instances of misuse across 145 projects. Notably, *Gopher* yields more pronounced benefits when applied to projects that are highly imported or starred.

The enhancements in *Gopher*'s accuracy are attributable to three main factors: (*a*) the constraint derivation function in our framework, (*b*) an accurate data flow tracking algorithm, and (*c*) a more

Table 2: Detection Results on 145 Small-scale Test Projects.

	Tools	Shared Rules		R19		New Rules		Total		
		$\#_{TP}$	$\#_{FP}$	$\#_{TP}$	$\#_{FP}$	$\#_{TP}$	$\#_{FP}$	$\#_{TP}$	$\#_{FP}$	$\#_{Dr}$
S_i	**Gopher**	298 (98%)	5	27	0	25	0	350	5	26
	CryptoGo	207 (78%)	58	8	0	0	0	215	58	–
S_s	**Gopher**	448 (99%)	2	28	0	42	2	518	4	20
	CryptoGo	321 (76%)	102	2	0	0	0	323	102	–
S_o	**Gopher**	123 (98%)	2	17	0	24	0	164	2	3
	CryptoGo	85 (76%)	31	6	0	0	0	91	31	–

S_i, S_s, and S_o: the 145 test projects selected from Go projects that are highly-imported, highly-starred, and of ordinary categories.
Shared Rules: the rules defined by both tools (13 rules: R03, R05-R13, R15-R17).
New Rules: the new rules introduced by *Gopher* (5 rules: R01, R02, R04, R14, R18).
$\#_{TP}$, $\#_{FP}$: the number of true positives, false positives in a tool's output.
$\#_{Dr}$: the number of extra misuses discovered by the constraint derivation feature.

exhaustive set of cryptographic detection rules. Improvements (*a*) and (*c*) occur within *CryDict*, while (*b*) is applied to the implementation of *Detector*. The logical decoupling enables the former two to be efficiently adapted to other detection tools.

(1) *Our framework's novel CryDict constraint derivation function* is pivotal, particularly for high-imported projects, contributing to a 19% improvement in accuracy and an average detection of 0.52 misuses per project, highlighting its significant impact on widely-used community projects.

It is noteworthy that across the 145 test projects, a total of 986 actual misuses were identified. Excluding the misuses (525 instances) related to the dangerous or warning functions designated by R05, R06, and R19, there were a total of 461 recorded instances of other types of misuses. Among these, misuses triggered by cross-library calls account for 9.10% (42 out of 461) and represent 3.75% of the overall cross-library API calls in these projects (42 out of 1121). This indicates that while encapsulating cryptographic APIs to simplify their use can significantly reduce the likelihood of misuse, the potential for cryptographic misuse arising from cross-library interactions remains non-negligible and warrants attention.

Table 3: Breakdown of Accuracy on Small-scale Test Projects.

Rules	# of Projects	# of Alerts	$\#_{TP}$	$\#_{Dr}$
R01	4 (2.76%)	9	9 (100%)	1 (11.11%)
R02	5 (3.45%)	6	6 (100%)	0 (0.00%)
R03	30 (20.69%)	79	77 (97.47%)	10 (12.66%)
R04	31 (21.38%)	57	55 (96.5%)	2 (3.51%)
R05*	84 (57.93%)	415	415 (100%)	N/A
R06*	19 (13.10%)	40	40 (100%)	N/A
R07	1 (0.69%)	1	1 (100%)	1 (100.00%)
R08	8 (5.52%)	8	7 (87.5%)	1 (12.50%)
R09	12 (8.28%)	19	17 (89.4%)	1 (5.88%)
R10	3 (2.07%)	5	5 (100%)	2 (40.00%)
R11	1 (0.69%)	2	2 (100%)	2 (100.00%)
R12	22 (15.17%)	82	82 (100%)	12 (14.63%)
R13	52 (35.86%)	148	144 (97.3%)	10 (6.76%)
R14	8 (5.52%)	10	10 (100%)	0 (0.00%)
R15	2 (1.38%)	21	21 (100%)	0 (0.00%)
R16	0	0	N/A	N/A
R17	7 (4.83%)	20	20 (100%)	0 (0.00%)
R18	1 (0.69%)	5	5 (100%)	0 (0.00%)
R19*	26 (17.93%)	70	70 (100%)	N/A
Total	**N/A**	**997**	**986 (98.9%)**	**42**

$\#_{TP}$: the number of true positives *Gopher*'s output.
$\#_{Dr}$: the number of misuses discovered by the constraint rule derivation feature.
*: the constraint derivation is disabled for these rules to avoid unnecessary alerts.

Table 4: Breakdown of the Reduction of False Positives in 145 Small-scale Test Projects via Four Optimization Strategies.

Method	Num	Percentage	Method	Num	Percentage
OP1	162	36.40%	OP3	25	5.62%
OP2	226	50.79%	OP4	32	7.19%
Total	**445**				

(2) *Our Detector's accurate data flow tracking algorithm* is also a key factor in enhancing accuracy. This includes a more nuanced analysis of program calls, the treatment of built-in functions within Go's implementation, and a flow-, context- and field-sensitive analysis. These advancements have led to the identification of 207 true misuses, representing a 33.7% improvement in performance across the 145 projects evaluated.

(3) *The expanded cryptographic detection rules* have significantly bolstered our tool's performance, particularly for high-star projects, where each project, on average, revealed 0.84 additional misuses through the application of these new rules. These APIs are widely utilized, often involve the misuse of password hashing, an area that has been largely overlooked in previous research.

Breakdown. Table 3 provides a detailed breakdown of the accuracy of cryptographic misuse detection for our tool across the 145 test cases. It can be observed that our tool demonstrates a commendable level of accuracy regardless of the specific rule being applied. As outlined in Section 5.4, our optimization methods, which are tailored to the Go language and cryptographic APIs, have significantly contributed to reducing false positives. As detailed in Table 4 , these methods have helped us eliminate 445 false alarms across 145 projects, achieving a remarkable reduction in the false-positive rate by 97.5%. We have manually verified that all removed alerts were indeed false positives. With these optimization methods, *Gopher* was able to detect an additional 64.1% of misuses in the test projects, all while maintaining an accuracy rate of 98.9%. However, the high rate of false positives in *CryptoGo* can be largely attributed to its less precise cross-library analysis and a deficiency the program's implementation phase.

6.2.2 Performance Comparison. When evaluating runtime performance, we conducted a comparison of the execution times between *CryptoGo* and *Gopher* across 145 test projects. *CryptoGo* scanned the projects with an average time of 47.1 seconds, while *Gopher* took an average of 51.5 seconds, showcasing a commendable scan rate of 10,941 lines/sec. Compared to *CryptoGo*, we trade 9% extra time for 64.1% more detected misuses and an additional 25.9% cross library API coverage. This performance can be attributed to our implementation of an optimized sparse data flow analysis, as detailed in Section 5.2, in conjunction with multi-threading enhancements. It should be noted that constraint derivation and misuse detection can be conducted concurrently, thus rendering the overhead introduced by the former negligible. Furthermore, when performing cross-library detection, *Gopher* only needs to load the *CryDict* constraint documents corresponding to the target functions of the called libraries. There is no need to reconstruct all the data flow information involved in the complete cross-library call path, thus significantly reducing time and memory overhead.

Table 5: Distribution of Misuse in Go Ecosystem.

Rules	# of Alerts		$\#_{Dr}$	Rules	# of Alerts		$\#_{Dr}$
	Total	H-Star			Total	H-Star	
R01	376	36	24 (6.38%)	R11	335	23	71 (21.19%)
R02	987	39	48 (4.87%)	R12	3,639	653	360 (9.89%)
R03	2,775	295	158 (5.70%)	R13	5,522	790	112 (2.03%)
R04	5,929	388	206 (3.47%)	R14	1,278	110	0 (0.00%)
R05*	18,668	1,705	N/A	R15	725	107	0 (0.00%)
R06*	2,207	145	N/A	R16	8	0	0 (0.00%)
R07	52	6	11 (21.15%)	R17	574	84	15 (2.61%)
R08	1,011	53	205 (20.28%)	R18	196	18	0 (0.00%)
R09	3,136	167	499 (15.92%)	R19*	6,236	402	N/A
R10	503	9	36 (7.16%)	**Total**	**54,157**	5,027	**1,745**

H-Star: highly-stared projects (with over 1,000 GitHub stars).
*: the constraint derivation is disabled for these rules to avoid unnecessary alerts.

6.2.3 Integration of CryDict with CryptoGo. To substantiate the *tool-independent* nature of *Crydict*, we enhanced *CryptoGo* by modifying its existing key-length taint tracker. Initially, this tracker relied on hard-coded constraints; we updated it to utilize *Crydict* constraints for cross-library analysis of the representatives R03 and R04, demonstrating the compatibility of *Crydict*.

(1) Enhancement of Existing Misuse Detection Scenarios. The original *CryptoGo* was already capable of identifying misuses of type R03, detecting 49 instances within our test projects. By integrating *CryDict*, the detection of R03 misuses increased to 77, a significant rise of 57.1%. This improvement stems from two key factors: the heavy influence of encapsulation on R03 scenarios and the original *CryptoGo*'s limitations in cross-function and cross-library analysis, which were significantly mitigated through *CryDict* integration.

(2) Expansion of Detectable Misuse Scenarios. CryptoGo was initially unable to detect misuses related to R04. However, with *CryDict* as the constraint source, it successfully identified 50 instances of R04 misuse. This illustrates the flexibility and expanded detection scope achievable with dynamic *CryDict* constraints, effectively broadening the horizons of existing detectors.

The integration not only deepened the detection capabilities for known misuse rules but also extended the reach to new misuse scenarios. This underscores *CryDict*'s potential to enrich static analysis tools in the domain of cryptographic misuse detection. Future efforts will explore integrating *CryDict* with tools in other programming languages.

6.3 Security Insights within the Go Ecosystem

6.3.1 General Assessment of Cryptographic Misuse within the Go Ecosystem. We conducted a comprehensive examination of 19,313 Go projects, identifying 54,147 instances of misuse across as many as 13,000 projects (67.31%), with an average of 2.8 misuse instances per project. Table 5 provides a detailed breakdown of the distribution of these vulnerabilities across the various rules examined.

The most prevalent issues include misuses related to R05 (dangerous algorithms), R19 (deprecated functions), R04 (low iteration in password hash), and R13 (not verify certificates in TLS), each with over 5,000 occurrences. The most critical issue is the misuse of outdated and potentially dangerous algorithms, which is found in 33% of Go cryptography projects.

We analyzed the frequency with which each type of misuse occurs within individual projects. Most types of misuse appear

sporadically; however, some misuses are repeatedly observed, indicating systemic issues within projects for these types, such as R04, R05, and R13, for details, see Appendix D.2. We further explored the relationship between project popularity and misuse distribution in Appendix D.3. We also identified that most rules have little correlation with each other. However, there are indeed some rules that do affect each other's misuse patterns, including unidirectional and bidirectional influences, see Appendix D.4 for details.

6.3.2 Influence of Constraint Derivation on Detection Effectiveness. Excluding misuses related to R05, R06, and R19 (27,111 instances), there are a total of 27,046 instances of other types of misuses, among which 6.45% (1,745 out of 27,046) were detected through *CryDict*-based constraint derivation. The influence of constraint derivation on the spectrum of misuse types is notably diverse. As shown in Table 5, the rules that are most profoundly influenced by the constraint derivation component are R08 (constant salt), R09 (constant key), and R11 (constant vector). Approximately 20% of hard-coded cryptographic issues necessitate a detailed cross-library examination (based on *CryDict*-based constraint derivation) to be identified. This indicates a heightened likelihood for symmetric encryption and authentication algorithms to be incorrectly utilized once encapsulated across various libraries.

Moreover, the enhancements in detection coverage from constraints deduction, as seen in Table 2, are notably more pronounced in projects with high import counts and star ratings, exceeding that of randomly chosen projects by a factor of more than five. This underscores the tendency of widely-adopted projects to encapsulate cryptographic APIs, prioritizing usability. Performing cross-library analysis on such influential projects markedly broadens the coverage of misuse detection.

6.3.3 Representative Security Findings. Our in-depth security evaluation unveiled significant vulnerabilities within the Go ecosystem. Notably, we identified critical security lapses in widely-adopted projects, which, despite their popularity, exhibited a concerning tendency to prioritize convenience over security protocols. It should be noted that we have reported all these findings to the developers, and have received their acknowledgment and commitment to fix them as soon as possible. Table 6 displays some projects that have been assigned CVE IDs. It can be seen that it involves various types of projects, including web frameworks, single sign-on (SSO), and multi-factor authentication (MFA) that are related to security operations. These projects are widely popular (with over 8,000 GitHub stars), so cryptographic misuse could lead to more extensive and serious consequences.

Table 6: Partial Security Findings with Acknowledged, Fixed and Assigned CVE IDs.

CVE-ID	CVSS Score	Misuse Rules	Project Type	Project Popularity (#*Github Stars*)
CVE-2024-40464 CVE-2024-40465	8.8 8.8	R13 R05	Web Framework	31,324
CVE-2024-41253	7.1	R13	Web Framework	11,260
CVE-2024-41259	9.1	R05	Music Server	11,020
CVE-2024-41260	7.1	R10	SSO, MFA	10,271

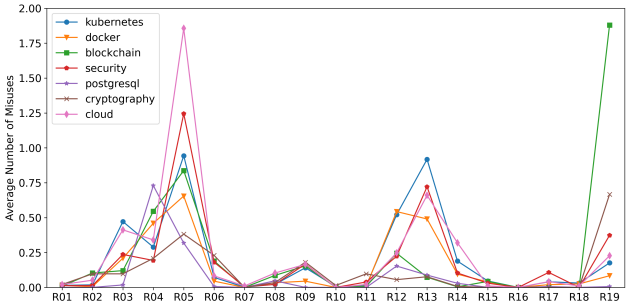


Figure 10: Characteristics of Misuse Distribution Across Application Types.

One such instance involved a favored web framework, boasting over 30,000 GitHub Stars and deployment across several major internet corporations. We discovered that this framework systematically bypasses TLS verification for SMTP client communications, omitting additional security checks. Although this simplifies development, it exposes the framework to Man-in-the-Middle (MitM) attacks, thereby jeopardizing the confidentiality and integrity of email exchanges.

Similarly, in a SSO platform with over 9,000 GitHub Stars, we discovered that TLS authentication is forgone during client proxy configurations. The absence of proxy service's TLS certificate authentication opens the door for attackers to eavesdrop on, tamper with, or forge communications between clients and servers, thereby undermining the security of SSO services [68].

In another project offering remote access, with a notable 10,000 GitHub Stars, we identified hard-coded IV usage in CBC mode encryption during database interactions. Such a practice could render the service susceptible to selective plaintext attacks, potentially exposing the distribution of plaintext data [68].

Moreover, in a messaging service with over 5,000 GitHub stars, the server configuration was set to support TLS 1.0 as the minimum version. This opens up the possibility for attackers to downgrade the server maliciously and launch *BEAST* attacks, leading to session key leakage and impersonation of server pushes or eavesdropping on conversations [49].

Therefore, developers of popular projects should prioritize comprehensive security practices to safeguard users who may not possess an in-depth understanding of cryptography. It is imperative to maintain the reliability of the software development community.

6.3.4 Characteristics of Misuse Distribution Across Application Types. In our examination, we also focused on seven prevalent categories within Go development, including Kubernetes for web services, Blockchain, and cloud-based applications, etc. To ensure a comprehensive representation, we included a minimum of 100 projects per application type to delineate their unique characteristics and specific misuse scenarios, as depicted in Figure 10.

For instance, *cloud* applications frequently exhibit issues related to the use of outdated cryptographic algorithms (R05), emphasizing the need for regular algorithm updates. In the realm of *Docker* and *Kubernetes*, which streamline deployment and management of services, we observed a higher incidence of improper HTTP connection handling, potentially due to configuration oversights

in the pursuit of operational efficiency. *SQL* applications, often involving user authentication, commonly struggle with inadequate password hashing iterations (R04), a problem that can be mitigated by adhering to recommended security practices.

Within *blockchain* and *cryptography* applications, there was an alarmingly high utilization of deprecated functions (R19), possibly due to the engagement with lower-level cryptographic operations where alternative functions are either non-existent or impractical. For instance, elliptic.Marshal and elliptic.Unmarshal, which are used to convert points on the elliptic curve and byte encoding as per the SEC 1 standard [13], are frequently misused, accounting for 69% of all deprecated function usage[1]. Despite the Go documentation advocating for crypto/ecdh as a substitute, its encoding method proves less versatile for elliptic curve operations, especially in blockchain applications where the ecdh protocol is not typically employed alongside elliptic curve encoding.

Interestingly, applications designated as "*security*" type, which one might assume to be more secure, actually exhibit a higher rate of rule violations, including R03, R05, R06, R09, R13, R17, and R19. These applications encompass control over physical access to hardware as well as safeguards against attacks originating from network access, data injection, and code injection. On average, each *security* type project averages 3.7 instances of misuse, surpassing the overall average of 2.8 instances, indicating that these applications may require a more vigilant approach to cryptographic practices.

7 DISCUSSION

7.1 Limitations

While our efforts to minimize false positives and negatives have been extensive, several limitations have been identified that may inadvertently lead to such occurrences.

(1) False Positives. As illustrated in Section 6.2.1, during the comprehensive small-scale testing of *Gopher*, which included 145 projects, we conducted a detailed manual analysis that identified 11 false positives. Sources of false positives in our framework can be categorized into three main types, with the respective numbers of false positives being 0, 10, and 1. Notably, the first category is introduced by our novel approach, while the remaining types are prevalent in existing tools.

(a) Expressiveness of CryDict ($\#_{FP}$ = 0). As described in Section 4.6, *CryDict* has deliberately simplified its expressive power to maintain the clarity of constraints. However, this has inevitably led to challenges in constraint description in a very small number of scenarios, resulting in false positives, such as when encountering functions like *scrypt* (see Section 4.6 for details). It should be noted that such false positives are exceedingly rare. In our rigorous analysis and validation of misuse reports from 145 projects, no false positives were attributed to this particular flaw. This indicates that the risk of false positives introduced by *CryDict* is negligible.

(b) Path Insensitivity ($\#_{FP}$ = 10). Path insensitivity is a prominent source of false positives in our 145 testing projects. Seven false positives stem from the oversight of conditional logic ("`if`" and "`switch`"), that incorrectly evaluate paths deemed implausible, such as setting the length of RSA key to zero. An additional three false positives arise from developers using "`for`" loops to populate key arrays with random values; path insensitivity triggers erroneous alerts on these arrays before their actual initialization.

(c) Call Graph Construction via CHA ($\#_{FP}$ = 1). Utilizing CHA to construct function call diagrams might also contribute to a slight increase in false positives. This occurred because both Ed25519 and RSA shared a common interface for key generation in a project, leading to misinterpretation in the mapping of function calls and, thus, false positives. Given the minimal incidence of false positives compared to the significant time costs associated with employing more complex strategies for charting program call graphs, this trade-off is justified.

(2) False Negatives. Our *Detector* specializes in analyzing Go language code, excluding external or config files in other languages, such as those involving cgo. This limited scope may restrict the detection of misuses tied to non-Go code elements.

(3) Constraint Derivation Scope. The current implementation of our framework is designed to automatically generate usage constraints for cryptographic libraries that extend the official Go cryptography library. However, it lacks support for libraries built from the ground up.

7.2 Responsible Disclosure and Open Source

Responsible Disclosure. In the spirit of transparency and to address security vulnerabilities, we proactively contacted developers whose projects were flagged for cryptographic misuse. As of August 2024, we have employed the second phase of our approach to conduct a more comprehensive and detailed disclosure for high-severity vulnerabilities involving projects with over 10,000 GitHub stars. The more information about disclosure is in Appendix E.

Open Source. We have open-sourced the *Gopher* instantiation project on GitHub, including executable files and example documentation[2]. We will regularly update our basic constraints to ensure alignment with the latest security standards and extend *Gopher* to accommodate other programming languages. Additionally, we will keep our disclosure progress and CVE application status updated on the homepage.

8 RELATED WORK

In the domain of cryptographic API misuse detection, a variety of approaches exist, predominantly featuring static and dynamic analysis methods. This section provides an overview of the current work and offers a comparative summary of the prevailing detection methods oriented toward different programming languages.

Static analysis. *CryptoLint* [20] extends the capabilities of *Androguard* [18] by employing inter-procedural backward slicing to assess cryptographic misuse of in Android. *CogniCrypto$_{SAST}$* [36] enforces manual rules and utilizes custom definition languages to separate detection rules from the tools themselves. *CryptoGuard* [49] significantly expands the scope of detection rules and enhance precision by designing a set of specialized refinement algorithms. *LICMA* [61] identifies common misuses in five Python crypto APIs. *TAINTCRYPT* [48] tailored for C/C++, introduces a paradigm for detecting implementation flaws at compile time using program

[1]The Go official documentation does not state the specific reasons for the deprecation of these two functions.

[2]Available at https://github.com/yxzhang2024/gopher.

Table 7: Tools for Detecting Cryptographic Misuse in Source Code Across Various Programming Languages.

Name	Method	Lang.	$\#_{Test}$	$\#_{Eco}$	$\#_{Rule}$	Rule Source	Scalable Range †	Cross-Library ‡	Context-, Flow-Sensitive ≀
Crylogger [46]	Dynamic	Java	150	1,780	26	Hard-coded	✗	✓	✗
Xu et al.'s tool [65]	HMM	Java	(3,953)	-	-	Training data	✓	✗	✗
CogniCrypt$_{SAST}$[36]	Static	Java	50	8,422	7	Manual	✓	✗	✓
CryptoGuard [49]	Static	Java	46	6,181	16	Hard-coded	✗	✗	✓
LICMA [61]	Static	Python	-	946	5	Hard-coded	✗	✗	✗
TAINTCRYPT [48]	Static	C/C++	5	-	(15)	Manual	✓	✗	✓
CryptoGo [39]	Static	Go	120	-	12	Hard-coded	✗	✗	✗
Gopher	Static	Go	145	19,313	19	Manual+Derived	✓	✓	✓

$\#_{Test}$: the number of test projects evaluated in-depth for misuse, where *Xu et al.'s tool* uses the analysis results from [36] that have not been fully verified as the benchmark.
$\#_{Eco}$: the number of test projects for ecosystem assessment or large-scale measurement.
$\#_{Rule}$: the number of misuse rules or types. *TAINTCRYPT* provided detection methods for 15 rules, but due to the diversity of APIs, they only evaluated 5 rules in the experiment.
†: the cryptographic API detection scope can expand without re-development; ‡: supports cross-library detection; ≀: the detection field-/flow-sensitive.

analysis techniques. *CryptoGo* [39] integrates the classification of algorithms into misuse detection for the first time, achieved misuse detection of Go. Distinct from the aforementioned studies, *CryptoREX* [67] introduces a scalable framework that detects cryptographic misuse across diverse IoT firmwares by lifting binary code to IR and performing static taint analysis. The study on 521 firmware images found that 24.2% had at least one cryptographic misuse, revealed the risks in the firmware.

Dynamic analysis. *SMV-Hunter* [55] and *AndroSSL* [25] focus on SSL/TLS detection, while *K-Hunt* [38] detects predictable keys by analyzing the execution of Java programs. *Crylogger* [46] includes a logger for monitoring cryptographic API usage and a checker for detecting incorrect API calls, significantly enhancing the performance of dynamic analysis.

Machine learning approaches. There has been a growing interest in using *machine learning* techniques, such as *Hidden Markov Models* (HMM). Studies have explored the use of AI models to identify patterns indicative of misuse [4, 23, 51, 65].

Evaluation research. The effectiveness of cryptographic misuse detection tools has been evaluated through various testing frameworks [15, 68]. *MASC* [3] is a mutation testing framework for cryptographic API misuse, provides a comprehensive evaluation of detection capabilities. These studies provided significant assistance in the implementation of our tool.

Table 7 provides a focused comparison of tools designed to cryptographic misuses in source code. Regarding the state of the cryptographic API application ecosystem, most current tools focused on the *Java* and have conducted misuse measurements on it at various scales of project sets, while there is a lack of large-scale measurement for the *Go*, as shown in $\#_{Eco}$ of Table 7.

These detection tools classify cryptographic API rules based on misuse scenarios and offer detection rules through various means, including hardcoding within the software, manual entry through external rule document, and semi-automatic extraction through training datasets. The origins of these rules dictate the tools' capability to expand their detection scope without re-development. For example, *Crylogger* has detailed 26 rules, the hard-coded approach makes it difficult to expand the detection scope without further development. *Xu et al.'s tool*, *CogniCrypto*$_{SAST}$, and *TAINTCRYPT* can update their rules by changing the training data or by manually updating external rule document. *Gopher* not only covers the rules addressed by these tools but further supports the updating through both manual and automatic derivation methods, achieving scalability in the detection scope of cryptographic APIs.

9 CONCLUSION AND FUTURE WORK

Our work introduces *Gopher*, an innovative framework designed to detect misuse of cryptographic APIs. *Gopher* integrates with *CryDict*, a concise and efficient method for constraint description and construction. This integration enables *Gopher* to autonomously update its constraint library during the detection process, based on base rules, and to explicitly articulate the implicit knowledge embedded within cryptographic libraries.

Through a comparative analysis with existing tools, we have demonstrated *Gopher*'s precision and adaptability. Our extensive analysis across 19,313 Go projects not only validated the practical utility of our framework but also revealed an additional 64.1% of misuse instances with a remarkable 98.9% accuracy rate. These findings have illuminated the prevalent cryptographic practices within the Go community, underscoring areas amenable to improvement.

The dynamic nature of cryptographic threats necessitates a detection framework that can evolve alongside these threats. In the future, we will use developer feedback from the ongoing responsible disclosure process to enhance our framework. We will focus on enhancing *CryDict*'s expressive ability for constraints, further refining *Gopher*'s ability to maintain a low false positive rate, and expanding its analytical scope and compatibility to ensure it can accommodate a more diverse array of ecosystems.

10 ACKNOWLEDGMENTS

The authors would like to thank anonymous CCS reviewers for their insightful suggestions and advice. This work was supported in part by the National Key R&D Program of China under Grant 2022YFB2701600; in part by the National Natural Science Foundation of China under Grant 62472021, 62202066, U21A20467, 61932011, and 62272457; in part by the Beijing Natural Science Foundation under Grant 4242023; and in part by the Youth Top Talent Support Program of Beihang University under Grant YWF-22-L-1272.

REFERENCES

[1] Yasemin Acar, Michael Backes, et al. 2017. Comparing the usability of cryptographic APIs. In *38th IEEE S&P*.
[2] Sriharitha Ambati. 2023. *Security and Authenticity of AI-generated code.* Ph. D. Dissertation. University of Saskatchewan.

[3] Amit Seal Ami, Nathan Cooper, et al. 2022. Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In *43th IEEE S&P*.

[4] Chunyan An, Donglei Zhang, et al. 2022. CryptoDetection: A Cryptography Misuse Detection Method Based on Bi-LSTM. In *2022 IEEE 8th ICCC*.

[5] Jean-Philippe Aumasson, Samuel Neves, et al. 2013. BLAKE2: simpler, smaller, fast as MD5. In *11th ACNS*.

[6] Elaine Barker and Quynh Dang. 2020. Nist special publication 800–57 part 1, revision 5: Recommendation for key management: Part 1-general, May 2020. *NIST, Tech. Rep* (2020).

[7] Karthikeyan Bhargavan and Gaëtan Leurent. 2016. On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *23rd ACM CCS*.

[8] Alex Biryukov, Daniel Dinu, et al. 2021. Argon2 memory-hard function for password hashing and proof-of-work applications. *IRTF* (2021).

[9] Daniel Bleichenbacher. 1998. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *18th CRYPTO*.

[10] Jenny Blessing, Michael A Specter, et al. 2024. Cryptography in the Wild: An Empirical Analysis of Vulnerabilities in Cryptographic Libraries. In *19th ACM AsiaCCS*.

[11] Fabrice Boudot, Pierrick Gaudry, et al. 2020. Comparing the difficulty of factorization and discrete logarithm: a 240-digit experiment. In *40th CRYPTO*.

[12] Cryptographic Mechanisms Bsi. 2020. Cryptographic Mechanisms: Recommendations and Key Lengths. *BSI—Technical Guideline* (2020).

[13] Certicom Research. [n. d.]. Standards for Efficient Cryptography 1 (SEC 1): Elliptic Curve Cryptography. Online at https://www.secg.org/sec1-v2.pdf.

[14] Mainak Chakraborty and Ajit Pratap Kundan. 2021. *Monitoring cloud-native applications: lead agile operations confidently using open source software*. Springer.

[15] Yikang Chen, Yibo Liu, et al. 2024. Towards Precise Reporting of Cryptographic Misuses. *31st NDSS* (2024).

[16] P David Coward. 1988. Symbolic execution systems—a review. *Software Engineering Journal* (1988).

[17] Gustavo Eloi de Paula Rodrigues, Alexandre M Braga, and Ricardo Dahab. 2023. Detecting cryptography misuses with machine learning: Graph embeddings, transfer learning and data augmentation in source code related tasks. *IEEE Transactions on Reliability* (2023).

[18] Anthony Desnos and Geoffroy Gueguen. 2018. Androguard documentation. *Obtenido de Androguard* (2018).

[19] Tim Düsterhus. 2023. *PHP RFC: bcrypt cost 2023*. PHP. https://wiki.php.net/rfc/bcrypt_cost_2023

[20] Manuel Egele, David Brumley, et al. 2013. An empirical study of cryptographic misuse in android applications. In *20th ACM CCS*.

[21] Tiago Espinha, Andy Zaidman, et al. 2014. Web API growing pains: Stories from client developers and their code. In *2014 CSMR-WCRE*.

[22] Sascha Fahl, Marian Harbach, et al. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *19th ACM CCS*.

[23] Felix Fischer et al. 2017. Stack overflow considered harmful? the impact of copy&paste on android application security. In *38th IEEE S&P*.

[24] Miles Frantz, Ya Xiao, et al. 2022. Poster: Precise detection of unprecedented python cryptographic misuses using on-demand analysis. In *28th NDSS*.

[25] François Gagnon, Marc-Antoine Ferland, et al. 2016. AndroSSL: A platform to test Android applications connection security. In *8th FPS 2015*.

[26] Martin Georgiev, Subodh Iyengar, et al. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *19th ACM CCS*.

[27] Mohammadreza Hazhirpasand, Mohammad Ghafari, and Oscar Nierstrasz. 2020. Java cryptography uses in the wild. In *14th ACM/IEEE ESEM*.

[28] Mohammadreza Hazhirpasand, Oscar Nierstrasz, et al. 2021. Hurdles for developers in cryptography. In *IEEE ICSME*.

[29] Chun-Ying Huang, Ching-Hsiang Chiu, et al. 2015. Code coverage measurement for Android dynamic analysis tools. In *International Conference on Mobile Services*.

[30] Atishay Jain. 2022. *Hugo in Action: Static Sites and Dynamic Jamstack Apps*. Simon and Schuster.

[31] JetBrains. [n. d.]. *Go Programming - The State of Developer Ecosystem in 2023*. https://www.jetbrains.com/lp/devecosystem-2023/go/

[32] Burt Kaliski. 2000. *PKCS# 5: Password-based cryptography specification version 2.0*. Technical Report.

[33] Hubert Kario. 2023. Everlasting ROBOT: The Marvin Attack. In *European Symposium on Research in Computer Security*.

[34] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. 1997. RFC2104: HMAC: Keyed-hashing for message authentication.

[35] Stefan Krüger, Sarah Nadi, Michael Reif, et al. 2017. Cognicrypt: Supporting developers in using cryptography. In *IEEE ASE*.

[36] Stefan Krüger, Johannes Späth, et al. 2019. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering* (2019).

[37] Himanshu Kumar, Sudhanshu Kumar, et al. 2013. Rainbow table to crack password using MD5 hashing algorithm. In *2013 IEEE Conference on Information & Communication Technologies*. IEEE.

[38] Juanru Li, Zhiqiang Lin, et al. 2018. K-Hunt: Pinpointing insecure cryptographic keys from execution traces. In *25th ACM CCS*.

[39] Wenqing Li, Shijie Jia, et al. 2022. Cryptogo: Automatic detection of go cryptographic api misuses. In *38th ACSAC*.

[40] Prerna Mahajan and Abhishek Sachdeva. 2013. A study of encryption algorithms AES, DES and RSA for security. *Global journal of computer science and technology* (2013).

[41] Na Meng, Stefan Nagy, et al. 2018. Secure coding practices in java: Challenges and vulnerabilities. In *40th ICSE*.

[42] Christopher Meyer, Juraj Somorovsky, et al. 2014. Revisiting {SSL/TLS} implementations: New bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium*.

[43] Sarah Nadi, Stefan Krüger, et al. 2016. Jumping through hoops: Why do Java developers struggle with cryptography APIs?. In *38th ICSE*.

[44] OWASP. 2021. Password Storage Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. Accessed: 2023-01-23.

[45] Markus Stenberg Peter Yee. [n. d.]. SSH Parameters. https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml. Accessed: 2024-03-28.

[46] Luca Piccolboni, Giuseppe Di Guglielmo, et al. 2021. Crylogger: Detecting crypto misuses dynamically. In *42nd IEEE S&P*.

[47] Niels Provos and David Mazieres. 1999. Bcrypt algorithm. In *USENIX*.

[48] Sazzadur Rahaman, Haipeng Cai, et al. 2021. From theory to code: identifying logical flaws in cryptographic implementations in C/C++. *IEEE TDSC* (2021).

[49] Sazzadur Rahaman, Ya Xiao, et al. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *26th ACM CCS*.

[50] SAM Rizvi, Syed Zeeshan Hussain, and Neeta Wadhwa. 2011. Performance analysis of AES and TwoFish encryption schemes. In *2011 International Conference on Communication Systems and Network Technologies*.

[51] Gustavo Eloi de P Rodrigues, Alexandre M Braga, and Ricardo Dahab. 2020. Using graph embeddings and machine learning to detect cryptography misuse in source code. In *19th IEEE ICMLA*.

[52] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *15th ACM SIGPLAN-SIGACT*.

[53] Shao Shuai, Dong Guowei, et al. 2014. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*.

[54] N Smart, M Abdalla, E Bjørstad, et al. 2018. Algorithms, key size and protocols report (2018). *ECRYPT—CSA, H2020-ICT-2014—Project* (2018).

[55] David Sounthiraraj, Zhiqiang Lin, et al. 2014. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *21st NDSS*.

[56] Durga Suresh. 2013. Introduction to the go programming language. *Journal of Computing Sciences in Colleges* (2013).

[57] Rebecca Taft, Irfan Sharif, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *2020 ACM SIGMOD*.

[58] Sergei Tikhomirov. 2018. Ethereum: state of knowledge and research perspectives. In *FPS 2017*.

[59] Meltem Sönmez Turan, Elaine Barker, et al. 2010. Recommendation for password-based key derivation. *NIST special publication* (2010).

[60] Xiaoyun Wang and Hongbo Yu. 2005. How to break MD5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*.

[61] Anna-Katharina Wickert, Lars Baumgärtner, et al. 2021. Python crypto misuses in the wild. In *15th ESEM*.

[62] Anna-Katharina Wickert, Lars Baumgärtner, et al. 2022. To fix or not to fix: a critical study of crypto-misuses in the wild. In *2022 IEEE TrustCom*. IEEE.

[63] Pieter Wuille et al. 2012. Hierarchical deterministic wallets. *Bitcoin Improvement Proposal (BIP)* 32 (2012).

[64] Baowen Xu, Ju Qian, Xiaofang Zhang, et al. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* (2005).

[65] Zhiwu Xu, Xiongya Hu, et al. 2020. Analyzing cryptographic api usages for android applications using hmm and n-gram. In *IEEE TASE*.

[66] Tatu Ylonen. 1996. SSH–secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*.

[67] Li Zhang, Jiongyi Chen, et al. 2019. CryptoREX: Large-scale analysis of cryptographic misuse in IoT devices. In *22nd RAID*.

[68] Ying Zhang, Md Mahir Asef Kabir, et al. 2022. Automatic detection of Java cryptographic API misuses: Are we there yet? *IEEE TSE* (2022).

A PREDICATES AND SEMANTICS IN CRYDICT

As mentioned in Section 4, *CryDict* is designed with a clear focus on preventing common misuse scenarios associated with cryptographic APIs. Each predicate in *CryDict* serves a specific purpose,

aiming to guide developers towards secure practices. Below are the misuse scenarios that *CryDict* predicates are crafted to address.

A.1 Misuse Scenarios Addressed by CryDict

We present each category of cryptographic rules and misuse scenarios. Additionally, to facilitate the assessment of the potential severity of the misuse of each rule, we adopt existing grading methods [49, 62] to prioritize misuse alerts: classifying their severity into three levels—high, medium, and low, based on (*i*) the potential benefits to the attacker and (*ii*) the difficulty of the attack [49].
(*a*) *High-severity (H)* vulnerabilities are characterized by their potential for remote exploitation and the significant benefits they offer to attackers. For instance, MitM attacks [22, 26] can be initiated without physical access and can yield substantial gains for the attacker. The susceptibility of MD5 and SHA1 hashes to rainbow table attacks [37] further elevates the severity, enabling attackers to effortlessly retrieve the pre-images of these hashes. Such vulnerabilities can be exploited to forge digital signatures or compromise message integrity. (*b*) *Medium-severity (M)* vulnerabilities involve scenarios where an attacker can compromise secrets but at a higher cost or with more dedication. While not immediately exploitable, these vulnerabilities significantly reduce the effort required for an attack, providing attackers with considerable advantages. For instance, Chosen Plaintext Attacks (CPA) and vulnerabilities from predictability, which can substantially undermine system security [39, 49]. (*c*) *Low-severity (L)* vulnerabilities demand significant effort to exploit and generally necessitate prior access to the system [49]. Brute force attacks, which require considerable computational resources, are an example of a low-severity scenario. The cost associated with brute force cracking is high, making it a less attractive option for attackers.
R-01 (low): The use of short salts can compromise password security. However, the high prerequisites for such an attack, such as the need to acquire stored password hashes [44], result in a low severity rating. Moreover, the introduction of longer salts, while beneficial for security, may marginally affect performance due to increased storage requirements and verification latency [8].
R-02 and R-03 (low): The employment of excessively short keys, such as 64-bit keys in HMAC, or algorithms with minuscule sizes like RSA-512, can impact integrity and confidentiality [6, 12, 54]. The effort required for brute force cracking leads to a low severity rating. However, on the other hand, the rectification of these issues is straightforward, and considering the small cost of remediation relative to the exponential security benefits, they are recommended for fix despite their low severity.
R-04 (low): The low iteration parameters in password hashes become more vulnerable to brute force attacks only when an attacker gains access to the stored password hashes [44]. Given that such access is necessary and the cost of conducting brute force attacks remains high, the severity of this misuse is considered low.
R-05 (high): The deployment of vulnerable cryptographic algorithms [7] like DES and MD5 are prohibited due to their potential to be cracked by modern computational power [6, 12, 54]. Moreover, the availability of commercial rainbow tables allow attackers to easily obtain pre-images of MD5 hashes for typical passwords, leading to a high severity rating [37, 49].
R-06 (low): The use of warning cryptographic algorithms such as 3DES and SHA-224 is discouraged, but their resilience to brute force attacks and the substantial costs associated with replacement result in a low severity rating [6, 54].
R-07 (medium): The predictability of PRNGs is a significant source of vulnerabilities [46, 68], providing attackers with substantial advantages, but still requiring effort to exploit, thus earning a medium severity rating [49].
R-08 (low): The lack of random salts in password hashes presents an exploitable vulnerability; yet, the prerequisites for such attacks result in a low severity rating. Since a random salt is unique for every user, an attacker has to crack hashes one at a time using the respective salt rather than calculating a hash once and comparing it against every stored hash. This makes cracking large numbers of hashes significantly harder[44].
R-09 (high): The use of constant keys can lead to severe data leakage, impacting confidentiality and earning a high severity rating [68].
R-10 and R-11 (medium): The necessity for a randomized IV in CBC and CFB modes, along with the prohibition of a static IV in OFB/CTR/GCM modes, is a critical measure to safeguard against CPA [39, 49]. However, this method of cracking requires the attacker to have access to chosen plaintext, thus making the exploitation of IV misuse (predictable/constant) more costly compared to R09. Consequently, the additional steps required for exploitation result in a medium severity rating.
R-12 — R-18 (high): The use of obsolete TLS/SSL/SSH protocols and cryptographic suites can lead to MitM attacks [22, 26, 42], with a high severity rating due to their severe impact on system confidentiality and availability.
R-19 (low): Due to the vast and complex types of cryptographic algorithms involved, with varying degrees of harm, many deprecated functions may not pose immediate risks, and replacing some can be challenging, thus they are assigned low severity priorities. How, it is recommended to avoid using functions that have been deprecated in the Go standard library, as these may carry various reasons for deprecation and could pose security risks.

A.2 Predicates Utilized in CryDict

CryDict employs a set of predicates to define the constraints on cryptographic API usage, ensuring developers adhere to secure coding practices. Each predicate targets a specific misuse scenario, providing a mechanism to express and enforce security requirements. The predicates are designed to be simple yet expressive, allowing for clear communication of security expectations to developers. Table 8 provides a detailed description of the predicates used in CryDict, their types, meanings, and the specific misuse scenarios they address.

A.3 Navigating Infinite Loops in Symbolic Execution

Symbolic execution often faces the challenge of infinite loops, especially when path conditions are not considered. To mitigate this, *CryDict* imposes a limit of two iterations per node during the slicing process. Given that cryptographic parameters are generally less complex to process, this approach balances thorough analysis

Table 8: The Predicates Used in CryDict.

Predicate Name	Type	Meaning	Misuse Scenario
BYTE_LENGTH	Binary	Defines the minimum length of a byte slice.	Incorrect parameter configuration
EQ_FALSE	Unary	Specifies that the value must be false.	Skipping verification in SSH/TLS
GEQ	Binary	Sets a minimum value for a number parameter.	Small parameter values
HTTPS	Unary	Requires the use of HTTPS connections.	HTTP connections
RANDOM_BYTES	Unary	Cryptographically secure random byte slices.	Predictable key/IV
RANDOM_IO	Unary	Cryptographically secure random source.	Use of insecure PRNG
NOT_CONST	Unary	Prohibits constant values for slices.	Use of constant IV in OFB/CTR/GCM
SECURE_HMAC_HASH	Unary	Indicates secure hash for HMAC.	Weak algorithms, e.g., HMAC-MD5
SECURE_TLS_SUITE	Unary	Specifies the use of secure TLS suites.	Use of insecure TLS suites
SECURE_SSH_SUITE	Unary	Specifies the use of secure SSH suites.	Use of insecure SSH suites

with practical limitations, ensuring efficiency and precision in the majority of cases.

B USAGE RULES FOR CRYPTOGRAPHIC ALGORITHMS

B.1 Classification of Cryptographic Algorithms

The integrity of encrypted data is heavily dependent on the choice of cryptographic algorithms. Developers must navigate a landscape that includes a variety of algorithms, some of which may be inherently weak, such as MD5. Over time, the vulnerabilities of these algorithms can increase due to evolving attack strategies and advancements in computational power. It is crucial to differentiate between the intrinsic strength of an algorithm and its parameter configurations. For instance, RSA remains secure with sufficiently large parameters. We have classified the algorithms into three categories based on NIST-SP 800-57 [6]:

(1) Dangerous Algorithms: As shown in Table 9, this category includes algorithms that are potentially hazardous due to their maximum security strength being 80 bits or less, such as SHA1 and MD5, or those susceptible to known attacks, like *Blowfish* in symmetric encryption [7]).

(2) Warning Algorithms: These are algorithms that, while currently usable, should be replaced as soon as possible. They include algorithms with a maximum security strength of 112 bits, such as 3DES, as shown in Table 9.

(3) Recommended Algorithms: This category includes algorithms that, at present, show no significant security issues and are considered suitable for future use.

Table 9: Dangerous and Warning Algorithms.

Category	Algorithm	Reason	Type
Dangerous	RC4	Security strength ≤ 80	Symmetric
	DES	Security strength ≤ 80	Symmetric
	Blowfish	Birthday attacks[7, 50]	Symmetric
	TEA	Birthday attacks[7]	Symmetric
	XTEA	Birthday attacks[7]	Symmetric
	MD4	Security strength ≤ 80	Hash
	MD5	Security strength ≤ 80	Hash
	SHA1	Security strength ≤ 80	Hash
	Ripemd160	Security strength ≤ 80	Hash
	PKCS1V1.5	Robot/Marvin attack[9, 33]	Padding
Warning	3TDEA	Security strength = 112	Symmetric
	P224	Security strength = 112	Elliptic-curve
	SHA2-224	Security strength = 112	Hash
	SHA2-512/224	Security strength = 112	Hash
	SHA3-224	Security strength = 112	Hash
	HMAC-MD5	Security strength = 112	MAC

Table 10: Parameters Recommends.

Algorithm	Recommended Parameters	Ref
DSA	ParameterSizes=3072	[6, 54]
RSA	ParameterSizes=3072	[6, 54]
symmetric encryption(e.g.AES)	key=128-bit	[6, 54] [12]
HMAC	key=128-bit	[34]
Argon2i(d)	t=1, p=4 m=64*1024 (64MiB), salt=128-bit	[6, 54]
	t=3, p=4 m=32*1024 (64MiB), salt=128-bit	[12]
bcrypt	cost=12	[10]
blake2b	if used as hash: hash_size=32	[6, 54]
	if used as mac: hash_size=16,	[5, 12]
PBKDF2	HMAC-SHA1: iter=1,300,000, salt=128-bit	[44, 59]
	HMAC-SHA256: iter=600,000,	
	HMAC-SHA512: iter=210,000,	
scrypt	N=2^{17} (128MiB), r=8 (1KiB), p=1, salt=128-bit	[44, 59]
	N=2^{16} (64MiB), r=8 (1KiB), p=2, salt=128-bit	
	N=2^{15} (32MiB), r=8 (1KiB), p=3, salt=128-bit	
	N=2^{14} (16MiB), r=8 (1KiB), p=5, salt=128-bit	
	N=2^{13} (8MiB), r=8 (1KiB), p=10, salt=128-bit	

B.2 Recommended Parameters for Cryptographic APIs

Table 10 provides a summary of recommended parameters for commonly used cryptographic APIs, which are essential for maintaining security integrity. These parameters should be carefully reviewed in future analyses to ensure robust cryptographic practices.

Historically, much focus has been placed on the parameter settings of public key and hash algorithms, with less emphasis on other frequently used algorithms such as *Bcrypt*. *Bcrypt*, known for its effectiveness in password storage, requires careful configuration of its *cost* parameter. Each increment of the *cost* parameter effectively doubles the effort required to crack the password, necessitating regular updates to the parameters of the Cryptography API to reflect the increasing computational capabilities. Drawing from the default settings established in PHP updates from 2023 [19], we have set the recommended cost to 12. For instance, on a server featuring an Intel(R) E31245 @ 3.30GHz (2011 server) — calculating a single hash with a cost of 12 requires merely 0.25 seconds. This duration is negligible in the context of interactive login processes and does not impede user experience.

B.3 TLS and SSH Suites

The integration of cryptographic algorithms into TLS/SSL and SSH protocols is a critical component of secure communications. While TLS suites have been well-studied, SSH suites present unique considerations. Aligning with IANA's recommendations, we have established criteria for selecting secure SSH suites [45].

Suites incorporating CBC mode are deemed insecure due to their vulnerability to oracle padding attacks [46], and the inclusion of weak cryptographic algorithms like SHA-1 and RC4 further weakens security. Additionally, the practice of truncating Hash and MAC function outputs, such as reducing SHA-256 to 96 bits, can severely impact the collision resistance of the original function, thus compromising cryptographic integrity.

C PRECISION OPTIMIZATION METHODS

As mentioned in Section 5.4, to bolster the precision of our cryptographic misuse detection, we have implemented a series of optimization techniques. These methods are designed to mitigate the occurrence of false positives without compromising the integrity of detection.

C.1 OP2: Removal of Uninitialized Variable

Our analysis initially flagged uninitialized variables as potential sources of randomness, especially pertinent to predicates like RANDOM_BYTES and NOT_CONST.

Let's take the Ethereum code in Figure 11 as an example. The return value of the function concatKDF should satisfy the predicate RANDOM_BYTES. However, due to path insensitivity, the analyzer might assume a direct execution path from line 4-9, bypassing the for loop. This assumption could lead to the erroneous conclusion that the slice may not satisfy RANDOM_BYTES.

```
1  func concatKDF(hash hash.Hash,
2      z, s1 []byte, kdLen int) []byte {
3    counterBytes := make([]byte, 4)
4    k := make([]byte, 0, roundup(kdLen, hash.Size()))
5    for counter := uint32(1); len(k) < kdLen;
         counter++ {
6      ...
7      k = hash.Sum(k)
8    }
9    return k[:kdLen]
10 }
```

Figure 11: False Positives Caused by Path Insensitivity in Project Ethereum.

However, recognizing a common Go practice of memory allocation with make followed by value assignment, we have refined our approach. We now disregard warnings related to uninitialized memory (allocated with alloc or make) when a phi node is present in the path, unless it directly impacts the output's randomness. Although this could potentially lead to issues with randomness due to uninitialized memory, the likelihood of such occurrences in practical use is relatively low. This adjustment reduces false positives while maintaining vigilance against genuine security risks. Similarly, we disregard the influence of null values during the analysis of RANDOM_BYTES and NOT_CONST.

C.2 OP3: Contextual Analysis of Local HTTP Connections

The STRICT enforcement of HTTPS can inadvertently flag local HTTP connections as security lapses. Given the contained environment of local networks and the limited scope for external interception, we recalibrated our analysis. Our method now differentiates between local and external HTTP usage, reducing false positives in development and testing scenarios where HTTPS may not be mandatory, as setting up HTTPS requires the TLS certificate, which can add complexity to the development process. For instance, consider the following example:

```
resp, err := http.Get("http://localhost/" + filepath)
```

In this case, although the HTTP protocol is used, the communication does not traverse an external network, making it challenging for external attackers to directly intercept or tamper with the transmitted data. While there may be malicious users who can access the local network and potentially intercept or tamper with data sent via HTTP, for scenarios that do not handle sensitive data or are not exposed to public networks, the use of HTTP is deemed acceptable.

C.3 OP4: Special Considerations for HMAC as a KDF

While HMAC is primarily recognized for its role in authentication, wherein, the randomness of the HMAC key was of primary importance; its application as a Key Derivation Function (KDF) requires a nuanced approach.

In instances such as *BIP32* [63], where HMAC serves as a KDF, the randomness of the child key is inherently derived from the parent key, not the HMAC key itself. As shown in Figure 12, a real-world example, in this case, the randomness of the result primarily stems from the account, not the key parameter of HMAC. Consequently, we have adapted our detection criteria to dismiss warnings about key randomness and length in such cases, aligning with the functional requirements of a KDF.

```
1  func ReadTCPSession(user *protocol.MemoryUser,
       reader io.Reader) (*protocol.RequestHeader,
       buf.Reader, error) {
2    ...
3    hashkdf := hmac.New(func() hash.Hash { return
         sha256.New() }, []byte("SSBSKDF"))
4    hashkdf.Write(account.Key)
5    ...
6  }
```

Figure 12: KDF Based on HMAC in V2ray**.**

D MORE EXPERIMENTAL RESULTS

As mentioned in Section 6, our comprehensive experimentation has yielded further insights into the application and effectiveness of our cryptographic misuse detection methods. This section elaborates on the additional results that provide a deeper understanding of the security landscape within the Go ecosystem.

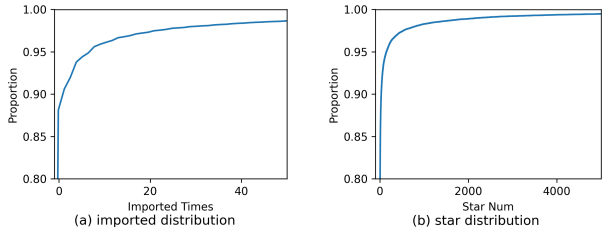


Figure 13: Imported and Star Distribution of All Go Projects.

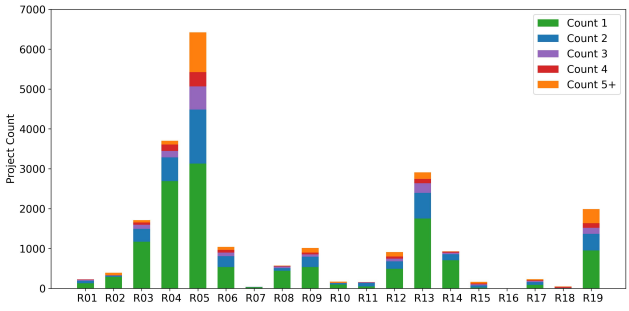


Figure 14: Distribution of Misuse Occurrence Times.

D.1 Basic Information of Scanned Project

The metrics of "imported by" count and GitHub "stars" serve as indicators of a package's popularity and influence. A high *imported by* count suggests widespread utility and quality, while a *higher star* count on GitHub reflects user appreciation and support, indicating a project's appeal and reliability within the open-source community. The distribution of these metrics across projects is illustrated in Figure 13. Interestingly, over 95% of projects have an "imported by" count of less than 10 and accumulate fewer than 500 GitHub stars. This statistic underscores the reality that the majority of users tend to concentrate on a select group of well-established projects, highlighting the dominance of these popular projects in the landscape of open-source software development.

D.2 The Distribution of Misuse Occurrence Times

An analysis of the frequency of misuse instances per project reveals that most misuse types appear sporadically. However, certain misuses, such as those related to R04 (Low Iteration Parameters), R05 (Dangerous Algorithms), R13 (TLS Certificate Verification Omission), and R19 (Deprecated Functions), have a higher probability of recurring multiple times within a project, indicating a systemic issue within projects. This reflects the lack of cryptographic knowledge and security awareness among developers, who may inadvertently and repeatedly use weak password algorithms or forego security verification to avoid the hassle of configuring certificates. This distribution is depicted in Figure 14. Within the R05 errors, MD5 is the most commonly utilized problematic algorithm, accounting for 52.34% of cases, followed by SHA1 (32.95%), and RC4 (5.73%).

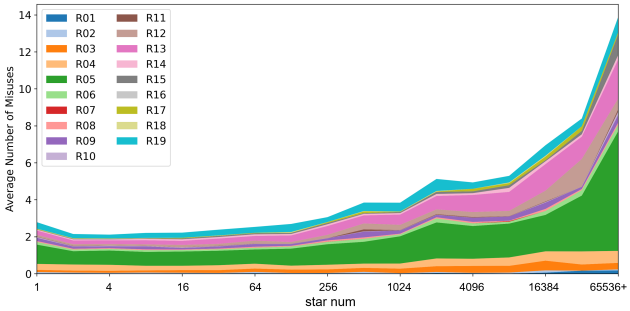


Figure 15: Relationship Between Project Popularity (GitHub Stars) and Misuse Distribution.

D.3 Relationship Between Project Popularity and Misuse Distribution

The adoption of cryptographic APIs in prominent projects carries significant weight, as they cater to a broader audience. With this comes an amplified responsibility for developers to uphold stringent security measures and prioritize robust default configurations to preempt misuse.

However, our research has uncovered a positive correlation between a project's popularity, as measured by GitHub stars, and the prevalence of cryptographic misuse. For example, in the 19,313 Go projects, there are 803 with more than 1,000 GitHub Stars, among which 684 exhibit misuse, involving a total of 5,027 instances of misuse, as shown in Table 5. Notably, projects with a higher star count tend to exhibit a greater number of misuse instances, particularly for misuse types R05 (Dangerous Algorithms) and R13 (Skip TLS Verification). This trend is illustrated in Figure 15, which suggests that increasing popularity and complexity in projects may inadvertently amplify the potential for security vulnerabilities.

D.4 Correlation Among Misuse Distributions Across Various Rules

In our investigation of the interplay among different rules' misuse distributions, we sought a quantitative approach to measure correlations. We introduced the concepts of information entropy H and mutual information I. Information entropy, as a measure of the quantity of information, and mutual information, to quantify the correlation between two events, were utilized to assess the relationship between different misuse occurrences.

We define the influence ability between the misuses of two different rules as: $IA(x, y) = \frac{I(x,y)}{H(y)}$, where $IA(x, y)$ represents the influence of rule x misuse on rule y misuse. If $IA = 1$, it indicates that the misuse information of rule y is entirely encompassed by rule x, suggesting that the misuse of x encompasses all the misuse information of y. Conversely, an influence ability of 0 (i.e., $IA = 0$) implies a negligible correlation between the two misuse types.

As shown in Figure 16, our analysis revealed that while 90.9% of rule pairs exhibit little correlation (with an influence ability of less than 0.05, i.e., $IA < 0.05$), certain rules do impact each other's misuse patterns.

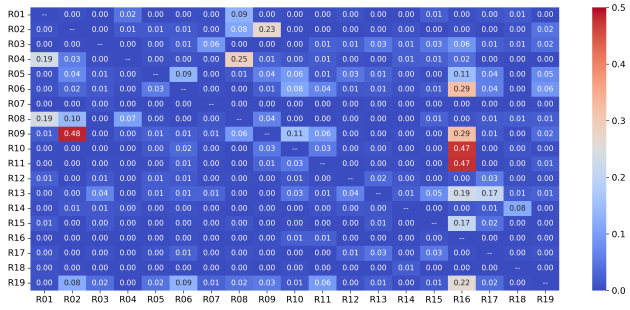


Figure 16: Inter-Rule Correlations in Misuse Patterns.

Unidirectional Influence. We identified a significant unidirectional influence where the misuse of one rule can predict the likelihood of others. Notably, the misuse of outdated TLS signature algorithms (R16) substantial impact on the occurrence of several other cryptographic issues, including the use of dangerous algorithms (R05), warning algorithms (R06), constant keys (R09), constant initialization vectors (R10 and R11), a lack of certificate verification in TLS (R13), and the use of deprecated functions (R19). Projects that exhibit R16 misuse are highly likely to struggle with these additional security concerns, indicating that R16 serves as a critical indicator of a project's cryptographic security posture. In fact, upon verification, we found that all projects with R16 misuse also presented with R05, R06, R09, R10, R11, R13, and R19 misuses, although the reverse was not always the case. The presence of R16 misuse is a red flag that may signal a broader range of security vulnerabilities within the project.

Bidirectional Influence. Our study also revealed instances where the misuse patterns exhibit a bidirectional correlation. This means that the presence of misuse in one rule can escalate the likelihood of misuse in another, and vice versa, creating a compounding effect on security vulnerabilities.

For example, there is a significant bidirectional influence between the use of keys that are too short (R02) and the misuse of constant keys (R09), as well as between the misuse of low iteration counts in password hashing (R04) and the use of constant salts (R08). This interplay suggests that projects which exhibit hardcoded keys or salts often face a concurrent struggle with selecting inappropriately low security parameters, indicating a more systemic issue with the security practices adopted within the development lifecycle of these projects.

E RESPONSIBLE DISCLOSE DETAILS

We proactively contacted developers whose projects were flagged for cryptographic misuse. Our goal was to responsibly disclose these vulnerabilities.

(a) Enhancing Response Rates. Due to the vast number of projects and instances of misuse, we initially reported the identified issues to developers using an automated, template-based approach. This includes information on the projects and the types and locations of the misuse identified. We observed a low engagement rate — only 24% developers responded. This prompted a strategic shift in our approach; After completing the aforementioned disclosures, we engaged in comprehensive dialogue with developers once again and assisted them in conducting a detailed analysis of the impact of misuse in their respective projects, including well-known ones such as `beego`, `hugo`, `pocketbase`, and `apache`. By offering a more nuanced analysis, including code corrections and references to library documentation, we increased the response rate to 57%, with 38% committing to or having already implemented fixes. This proactive strategy not only identified the misuse but also facilitated developers' understanding and commitment to rectifying the issues in future updates. As of August 2024, we have employed the second phase of our approach to conduct a more comprehensive and detailed disclosure for high-severity vulnerabilities involving projects with over 10,000 GitHub stars. We have been granted 16 CVEs to date.

(b) Developer Feedback. The positive feedback and acknowledgment we received have highlighted several key areas of misuse, often stemming from a lack of security awareness. For instance, developers have inadvertently employed MD5 for generating identifiers, overlooking its susceptibility to malicious exploitation [60]. This oversight can potentially lead to security breaches, compromising data integrity. Additionally, default settings that bypass TLS and SSH verification can expose systems to MitM attacks, enabling adversaries to intercept and alter communications. The use of outdated algorithms like SHA1 for certificate digests presents significant risks of certificate forgery [26]. Supporting outdated TLS versions for compatibility can also expose systems to known vulnerabilities, making them susceptible to attacks such as *POODLE* and *BEAST* [26]. Misinterpretations of cryptographic algorithms, such as using a constant IV in encryption, can enable cryptanalysis attacks, including chosen-plaintext attacks. Furthermore, low parameter settings in algorithms like PBKDF2 can diminish the computational effort required for brute-force attacks, potentially making password cracking more feasible.

(c) Challenges in Acknowledging Misuse. However, not all feedback was affirmative. Some developers were reluctant to concede to certain misuses, often due to the project's legacy or the presence of special scenarios. For instance, they argue that providing optional insecure configurations like TLS verification might be necessary for specific corporate environments. Some projects are ported from others, or they utilize protocols with inherent vulnerabilities. Updating these outdated algorithms unilaterally could result in severe compatibility issues. Consequently, developers may choose low-strength parameter configurations or older versions to pursue "perfect compatibility" or "high performance", such as opting for TLS 1.0 or use 1,000 iterations in PBKDF2, despite the recognized need for stronger security measures. Additionally, some legacy algorithms are not inherently insecure in all contexts, such as MD5 in hash tables, but the automatic detection of context-specific usage remains a common challenge for existing tools [39, 49].

After the completion of the disclosure, some developers, particularly those not specialized in cryptography, were unable to fully and directly comprehend the hazards brought about by misuses. Therefore, we responsibly engaged in the second phase of interaction, assisting them in analyzing the potential impacts of relevant misuses in their projects, and provided references to standard

specifications and authoritative recommendations. This detailed approach, though resource-intensive, has yielded positive outcomes. The process of this responsible follow-up interaction is ongoing.