CryptoGo: Automatic Detection of Go Cryptographic API Misuses

Wenqing Li, Shijie Jia, Limin Liu, Fangyu Zheng, Yuan Ma, Jingqiang Lin

ACSAC 2022
## Talk Outline

<table>
<thead>
<tr>
<th>Motivation</th>
<th>Rules</th>
<th>CryptoGo Design</th>
<th>Evaluation</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why did we start this work?</td>
<td>How to classify cryptographic algorithm and derive detection rules?</td>
<td>How does it work?</td>
<td>How is the performance?</td>
<td>Conclusions and reflections</td>
</tr>
</tbody>
</table>
Talk Outline

<table>
<thead>
<tr>
<th>Motivation</th>
<th>Rules</th>
<th>CryptoGo Design</th>
<th>Evaluation</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why did we start</td>
<td>How to classify cryptographic algorithm</td>
<td>How does it</td>
<td>How is the</td>
<td>Conclusions and reflections</td>
</tr>
<tr>
<td>this work?</td>
<td>and derive detection rules?</td>
<td>work?</td>
<td>performance?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What is cryptographic misuse?

- Incorrect implementations of cryptographic algorithms/protocols seriously jeopardize system security in practice.

- **Cryptographic Misuse**: The above erroneous implementations.
Cryptography can be used to offer basic security services (e.g., confidentiality, integrity, authenticity), thus constitutes the cornerstone of secure systems.

- Developers without cryptography knowledge
- Misusing various cryptographic APIs
- Reduce the security of cryptographic projects
Background

Issues

01 Are the cryptographic library APIs really secure?

- overly complicated
- poorly documented

02 Will developers really be able to make these cryptographic APIs work?

- lack cryptography experience
Cryptographic API misuses within the Go landscape are still uncovered.

Overview

In-depth analysis of the latest official Go cryptographic library (v1.18.3)

Tease out all the provided cryptographic algorithms

Put forward an algorithm classification method based on security strength and security vulnerability

Derive 12 cryptographic rules

Leverages static taint analysis technique

Takes a Go project program file as input and outputs a cryptographic misuse analysis report

1) Classify the cryptographic algorithms and derive the corresponding rules.

2) Develop a tool to detect cryptographic misuse issues in Go projects.

The two key insights
## Talk Outline

<table>
<thead>
<tr>
<th><strong>Motivation</strong></th>
<th><strong>Rules</strong></th>
<th><strong>CryptoGo Design</strong></th>
<th><strong>Evaluation</strong></th>
<th><strong>Conclusion</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Why did we start this work?</td>
<td>How to classify cryptographic algorithm and derive detection rules?</td>
<td>How does it work?</td>
<td>How is the performance?</td>
<td>Conclusions and reflections</td>
</tr>
<tr>
<td>Security Strength</td>
<td>Symmetric Key Algorithms</td>
<td>FFC (DSA, DH, MQV)</td>
<td>IFC* (RSA)</td>
<td>ECC* (ECDSA, EdDSA, DH, MQV)</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------</td>
<td>-------------------</td>
<td>-----------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>≤80</td>
<td>2TDEA</td>
<td>L=1024, N=160</td>
<td>k=1024</td>
<td>f=160-223</td>
</tr>
<tr>
<td>112</td>
<td>3TDEA</td>
<td>L=2048, N=224</td>
<td>k=2048</td>
<td>f=224-255</td>
</tr>
<tr>
<td>128</td>
<td>AES-128</td>
<td>L=3072, N=256</td>
<td>k=3072</td>
<td>f=256-383</td>
</tr>
<tr>
<td>192</td>
<td>AES-192</td>
<td>L=7680, N=384</td>
<td>k=7680</td>
<td>f=384-511</td>
</tr>
<tr>
<td>256</td>
<td>AES-256</td>
<td>L=15360, N=512</td>
<td>k=15360</td>
<td>f=512+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Security Strength</th>
<th>Digital Signatures and Other Applications Requiring Collision Resistance</th>
<th>HMAC, KMAC, Key Derivation Functions, Random Bit Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤80</td>
<td>SHA-1</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>SHA-224, SHA-512/224, SHA3-224</td>
<td>SHA-1, KMAC128</td>
</tr>
<tr>
<td>128</td>
<td>SHA-256, SHA-512/256, SHA3-256</td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>SHA-384, SHA3-384</td>
<td>SHA-224, SHA-512/224, SHA3-224</td>
</tr>
<tr>
<td>≥256</td>
<td>SHA-512, SHA3-512</td>
<td>SHA-256, SHA-512/256, SHA-384, SHA-512, SHA3-256, SHA3-512, KMAC256</td>
</tr>
</tbody>
</table>
Categorize All The Cryptographic Algorithms (NIST SP 800-57)

**Insecure cryptographic algorithms**
- the cryptographic algorithms which are with less than 112 bits security strength
- the cryptographic algorithms which have been broken into “insecure” cryptographic algorithms.
- the cryptographic algorithms which are disclosed to be vulnerable under specific scenarios

**Acceptable but not recommended cryptographic algorithms**
- the cryptographic algorithms with 112 bits security strength
- the cryptographic algorithms without secure vulnerability currently
  (they are currently considered to be secure through 2030, alternative algorithms which are more robust (e.g., ≥ 128 bits security strength) are commonly available.)

**Recommended cryptographic algorithms**
- the cryptographic algorithms which are with ≥ 128 bits security strength
- the cryptographic algorithms without secure vulnerability currently

**Security strength**: a number associated with the number of operations that is required to break a cryptographic algorithm or system.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Classification Type</th>
<th>Algorithm Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric-Key</td>
<td>Insecure</td>
<td>DES, 2TDEA, Blowfish, CAST5, TEA, XTEA, RC4</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Acceptable but not recommended</td>
<td>3TDEA, Twofish, Salsa20</td>
</tr>
<tr>
<td></td>
<td>Recommended</td>
<td>AES-128, AES-192, AES-256, ChaCha20-Poly1305</td>
</tr>
<tr>
<td>Asymmetric-Key</td>
<td>Insecure</td>
<td>RSA-512, RSA-1024, DSA-1024</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Acceptable but not recommended</td>
<td>RSA-2048, DSA-2048, ECDSA-P224</td>
</tr>
<tr>
<td></td>
<td>Recommended</td>
<td>RSA-3072, DSA-3072, ECDSA-P256, Ed25519, RSA-4096, RSA-7680, RSA-15360,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ECDSA-P384, ECDSA-P521</td>
</tr>
<tr>
<td>Hash Function</td>
<td>Insecure</td>
<td>MD4, MD5, SHA-1, RIPEMD-160</td>
</tr>
<tr>
<td></td>
<td>Acceptable but not recommended</td>
<td>SHA-224, SHA-512/224, SHA3-224</td>
</tr>
<tr>
<td></td>
<td>Recommended</td>
<td>SHA-256, SHA-512/256, SHA3-256, SHAKE-128, BLAKE2s, SHA-384, SHA3-384, SHA-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>512, SHA3-512, SHAKE-256, BLAKE2b</td>
</tr>
<tr>
<td>MAC Algorithm</td>
<td>Acceptable but not recommended</td>
<td>HMAC-MD5</td>
</tr>
<tr>
<td></td>
<td>Recommended</td>
<td>HMAC-SHA-1, Hash Functions (security strength ≥ 112 bits) based HMAC</td>
</tr>
<tr>
<td>ID</td>
<td>Rule Description</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>R-01</td>
<td>Do not use insecure cryptographic algorithms</td>
<td></td>
</tr>
<tr>
<td>R-02</td>
<td>Should use recommended algorithms preferentially</td>
<td></td>
</tr>
<tr>
<td>R-03</td>
<td>Do not use cryptographically insecure PRNG</td>
<td></td>
</tr>
<tr>
<td>R-04</td>
<td>Do not use predictable/constant cryptographic keys</td>
<td></td>
</tr>
<tr>
<td>R-05</td>
<td>Do not use the same password or salt for key derivation</td>
<td></td>
</tr>
<tr>
<td>R-06</td>
<td>IVs should be unique in CTR, OFB, GCM and XTS mode, and should be random in CBC and CFB mode</td>
<td></td>
</tr>
<tr>
<td>R-07</td>
<td>Do not use the padding PKCS#1-v1.5 for RSA</td>
<td></td>
</tr>
<tr>
<td>R-08</td>
<td>Do not use HTTP URL connections</td>
<td></td>
</tr>
<tr>
<td>R-09</td>
<td>Do not use weak SSL/TLS protocols</td>
<td></td>
</tr>
<tr>
<td>R-10</td>
<td>Do not use insecure cipher suites in SSL/TLS</td>
<td></td>
</tr>
<tr>
<td>R-11</td>
<td>Do not verify certificates or host names in SSL/TLS in trivial ways</td>
<td></td>
</tr>
<tr>
<td>R-12</td>
<td>Do not use insecure implementations deprecated by the official Go cryptographic library</td>
<td></td>
</tr>
<tr>
<td>Motivation</td>
<td>Rules</td>
<td>CryptoGo Design</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>Why did we start this work?</td>
<td>How to classify cryptographic algorithm and derive detection rules?</td>
<td>How does it work?</td>
</tr>
</tbody>
</table>
Design

The Analysis Procedure

Convert to Intermediate Representation

the input Go project program file $\rightarrow$ a static single-assignment (SSA) form intermediate representation (IR)

Taint Analyzer Construct

the well-targeted defined cryptographic rules $\rightarrow$ five kinds of taint analyzers

Taint Analysis

perform both backward and forward taint analysis
• A practical method of information flow analysis technology.
• Four types: sources, propagators, sanitizers/filters, and sinks.

**Taint Analyzer**

• A source function produces an untrusted input
• A sink function consumes an untrusted input sending it to a sensitive destination
• A propagator is a function that propagates the untrusted data from one point of the program (via a variable) to another
• A filter is a function that purifies an untrusted variable and makes it trustworthy
Taint Analyzer Construction

01 Insecure API Invocation Identification

Pattern Matching

- R-01 & R-02: Cryptographic Algorithms
- R-07: EncryptPKCS1v15; SignPKCS1v15
- R-12: curve25519.ScalarMult; bn256; pkcs12

The converted IR (SSA form) which is triggered several insecure APIs report the misuses/alerts
**Key Length Tracking**

Backward Taint Tracking (sink to source)

R-01 & R-02: Cryptographic Algorithms

```go
func ExampleRSA() {
    bits := 1024
    privateKey, err := rsa.GenerateKey(rand.Reader, bits)
    if err != nil { err }
    // The public key is a part of the *rsa.PrivateKey struct
    publicKey := privateKey.PublicKey
    // use the public and private keys
    // ...
}
```

---

Is the key length low?

several insecure APIs

backward taint analysis

filter

strong

low

report the misuses/alerts

(source)
the operation mode of block cipher (e.g., CBC, GCM);
the adopted elliptic curve of ECDSA (e.g., P224, P256);
the option of hash algorithm in HMAC (e.g., SHA-1, SHA-256).

\[ h := \text{hmac.New(md5.New, []byte (key))} \]
Randomness Tracking

**Backward Taint Tracking**
(sink to source)
R-03: Cryptographically Insecure PRNG
R-04: Predictable/constant Cryptographic Keys
R-05: Same Password or Salt
R-06: Predictable/constant IVs

```
func AESCBCEnc(plaintext []byte) (string, error) {
    key, _ := hex.DecodeString(s: "6368616e67652b077e69732b07617373")
    block, err := aes.NewCipher(key)
    if err != nil : err *
    ciphertext := make([]byte, aes.BlockSize+len(plaintext))
    iv := ciphertext[:aes.BlockSize]
    if _, err := ic.ReadFull(rand.Reader, iv); err != nil : err *
    blockMode := cipher.NewCBCDecrypter(block, iv)
    blockMode.CryptBlocks(ciphertext, plaintext)
    return base64.StdEncoding.EncodeToString(ciphertext), nil
}
```
Design

```go
def TLSTest() {
    config := tls.Config{
        InsecureSkipVerify: true,
        ServerName: "",
        CipherSuites: []uint16{
            tls.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
            tls.TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
            tls.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
            tls.TLS_RSA_WITH_AES_256_CBC_SHA,
        },
        MinVersion: tls.VersionTLS12,
    }
}
```

SSL/TLS Tracking

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-09</td>
<td>Weak SSL/TLS Protocols</td>
</tr>
<tr>
<td>R-10</td>
<td>Insecure Cipher Suites in SSL/TLS</td>
</tr>
<tr>
<td>R-11</td>
<td>Skip Certificate or Hostname Verification in SSL/TLS</td>
</tr>
</tbody>
</table>

Forward Taint Tracking
(source to sink)
- R-09: Weak SSL/TLS Protocols
- R-10: Insecure Cipher Suites in SSL/TLS
- R-11: Skip Certificate or Hostname Verification in SSL/TLS

**Is the SSL/TLS configuration correct?**

- **Correct**
- **Wrong**

(report the misuses/alerts)
<table>
<thead>
<tr>
<th><strong>Motivation</strong></th>
<th><strong>Rules</strong></th>
<th><strong>CryptoGo Design</strong></th>
<th><strong>Evaluation</strong></th>
<th><strong>Conclusion</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Why did we start this work?</td>
<td>How to classify cryptographic algorithm and derive detection rules?</td>
<td>How does it work?</td>
<td>How is the performance?</td>
<td>Conclusions and reflections</td>
</tr>
</tbody>
</table>
Implementation and Experimental Setup

• Around 2,236 lines of Go code to realize the construction of our taint analyzers

• Dataset from GitHub:
  • Sort by the most number of stars
  • Crawled 120 open source Go projects in total.
  • The average stars and forks are 5.9k and 562.62, respectively.
  • The maximum, minimum and average Line of Code (LoC) are around 1,128k, 0.2k and 152.04k, respectively.

• PC: Intel Xeon(R) E5-2682 v4 (2.50GHz CPU and 4GB RAM.)

• The average runtime: 86.27 milliseconds per thousand LoC.
Security Findings

- There are a total of 622 alerts for the 120 Go projects.
- Out of the 120 projects:
  - 100 projects (83.33%) have at least one cryptographic misuse
  - 73 projects (60.83%) have at least two misuses
  - 47 projects (39.17%) have at least three misuse
- Our careful manual source-code analysis confirms that 594 alerts are true positives, resulting in the accuracy as 95.50%.
- The 28 false positive cases are due to the path insensitivity, and the invocation of APIs from non-official Go cryptographic libraries
evaluation

misuse case | number of misuses
---|---
1. Insecure PRNG | 0
2. Predictable key in AES | 75
3. Reuse salt in HKDF | 8
4. Reuse salt in PBKDF2 | 18
5. Reuse salt in scrypt | 19
6. Not unique IV in CTR | 42
7. Not random IV in CBC | 31
8. Not random IV in CFB | 5

Total | 198

misuse case | number of misuses
---|---
1. TLS1.0 | 2
2. Insecure cipher suites | 5
3. Accept any certificate or host name | 34
4. HTTP URL | 115

Total | 156
Limitations

- CryptoGo may incur false negatives in the case of invocation of API from non-official Go cryptographic libraries (e.g., third-party cryptographic library or non-standard self-implemented cryptographic algorithms).
- The path insensitivity confuse the CryptoGo’s taint analyzers, which produces false positives.
- CryptoGo can only be done on a single application, and cannot perform inter-application analysis.
- CryptoGo can only cover the data stored in program files.

Disclosures

- Contacted 100 developers of the projects with cryptographic misuses/alerts and received email responses from 26 developers.
- 94 issues from 20 projects have been acknowledged and 33 issues from 6 projects have been declared as non-issue.
<table>
<thead>
<tr>
<th>Motivation</th>
<th>Rules</th>
<th>CryptoGo Design</th>
<th>Evaluation</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why did we start this work?</td>
<td>How to classify cryptographic algorithm and derive detection rules?</td>
<td>How does it work?</td>
<td>How is the performance?</td>
<td>Conclusions and reflections</td>
</tr>
</tbody>
</table>
• Introduce a static analysis tool CryptoGo, for detecting cryptographic API misuse in Go cryptographic projects.

• CryptoGo leverages static taint analysis technique, along with 12 cryptographic rules strongly coupled with Go cryptographic APIs and 5 kinds of specific taint analyzers.

• Implemented CryptoGo and carried out experiments based on 120 real-world Go cryptographic projects. CryptoGo identified 622 cryptographic API alerts (with an accuracy of 95.5%) and found that 83.33% of the Go cryptographic projects have at least one cryptographic misuse.
THANK YOU FOR WATCHING

Q & A

liwenqing@iie.ac.cn