CryptoGo: Automatic Detection of Go Cryptographic API Misuses

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

ACSAC 2022

中国科学院信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

中国科学技术大学 University of Science and Technology of China | 网络空间安全学院 School of Cyber Science and Technology

Talk Outline

Motivation	Rules	CryptoGo Design	Evaluation	Conclusion
Why did we start this work?	How to classify cryptographic algorithm and derive detection rules?	How does it work?	How is the performance?	Conclusions and reflections

Talk Outline

Motivation	Rules	CryptoGo Design	Evaluation	Conclusion
Why did we start this work?	How to classify cryptographic algorithm and derive detection rules?	How does it work?	How is the performance?	Conclusions and reflections

What is cryptographic misuse?

Cryptographic Misuse

“

- Incorrect implementations of cryptographic algorithms/protocols seriously jeopardize system security in practice.
- **Cryptographic Misuse:** The above erroneous implementations.

Background

“ 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)

↓ the Go standard library (i.e., crypto/...) and the supplemental repositories (i.e., golang.org/x/crypto/...)

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

the two key insights

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

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

Talk Outline

Motivation	Rules	CryptoGo Design	Evaluation	Conclusion
Why did we start this work?	How to classify cryptographic algorithm and derive detection rules?	How does it work?	How is the performance?	Conclusions and reflections

Classification

NIST SP 800-57

Security Strength	Symmetric Key Algorithms	FFC (DSA, DH, MQV)	IFC* (RSA)	ECC* (ECDSA, EdDSA, DH, MQV)
≤80 x	2TDEA	L=1024, N=160	k=1024	f=160-223
112 √ $\xrightarrow{2031}$ x	3TDEA	L=2048, N=224	k=2048	f=224-255
128	√ AES-128	L=3072, N=256	k=3072	f=256-383
192	AES-192	L=7680, N=384	k=7680	f=384-511
256	AES-256	L=15360, N=512	k=15360	f=512+

Security Strength	Digital Signatures and Other Applications Requiring Collision Resistance	HMAC, KMAC, Key Derivation Functions, Random Bit Generation
≤80 x	SHA-1	
112 √ $\xrightarrow{2031}$	x SHA-224, SHA-512/224, SHA3-224	
128	√ SHA-256, SHA-512/256, SHA3-256	SHA-1, KMAC128
192	SHA-384, SHA3-384	SHA-224, SHA-512/224, SHA3-224
≥256	SHA-512, SHA3-512	SHA-256, SHA-512/256, SHA-384, SHA-512, SHA3-256, SHA3-384, SHA3-512, KMAC256

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.

Classification

Algorithm	Classification Type	Algorithm Name
Symmetric-Key Algorithm	Insecure	DES, 2TDEA, Blowfish, CAST5, TEA, XTEA, RC4
	Acceptable but not recommended	3TDEA, Twofish, Salsa20
	Recommended	AES-128, AES-192, AES-256, ChaCha20-Poly1305
Asymmetric-Key Algorithm	Insecure	RSA-512, RSA-1024, DSA-1024
	Acceptable but not recommended	RSA-2048, DSA-2048, ECDSA-P224
	Recommended	RSA-3072, DSA-3072, ECDSA-P256, Ed25519, RSA-4096, RSA-7680, RSA-15360, ECDSA-P384, ECDSA-P521
Hash Function	Insecure	MD4, MD5, SHA-1, RIPEMD-160
	Acceptable but not recommended	SHA-224, SHA-512/224, SHA3-224
	Recommended	SHA-256, SHA-512/256, SHA3-256, SHAKE-128, BLAKE2s, SHA-384, SHA3-384, SHA-512, SHA3-512, SHAKE-256, BLAKE2b
MAC Algorithm	Acceptable but not recommended	HMAC-MD5
	Recommended	HMAC-SHA-1, Hash Functions (security strength ≥ 112 bits) based HMAC

Derive 12 Cryptographic Rules

ID	Rule Description
R-01	Do not use insecure cryptographic algorithms
R-02	Should use recommended algorithms preferentially
R-03	Do not use cryptographically insecure PRNG
R-04	Do not use predictable/constant cryptographic keys
R-05	Do not use the same password or salt for key derivation
R-06	IVs should be unique in CTR, OFB, GCM and XTS mode, and should be random in CBC and CFB mode
R-07	Do not use the padding PKCS#1-v1.5 for RSA
R-08	Do not use HTTP URL connections
R-09	Do not use weak SSL/TLS protocols
R-10	Do not use insecure cipher suites in SSL/TLS
R-11	Do not verify certificates or host names in SSL/TLS in trivial ways
R-12	Do not use insecure implementations deprecated by the official Go cryptographic library

Talk Outline

Motivation	Rules	CryptoGo Design	Evaluation	Conclusion
Why did we start this work?	How to classify cryptographic algorithm and derive detection rules?	How does it work?	How is the performance?	Conclusions and reflections

The Analysis Procedure

Convert to Intermediate Representation

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

Taint Analyzer Construct

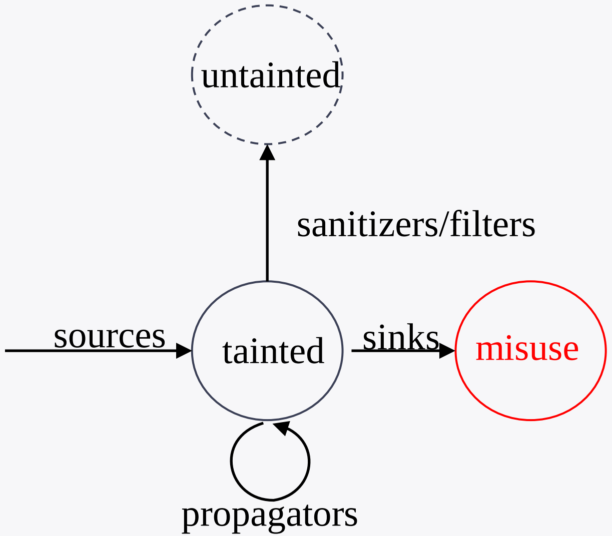
the well-targeted defined cryptographic rules → five kinds of taint analyzers

Taint Analysis

perform both backward and forward taint analysis

Taint Analysis

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: $curve25519.ScalarMult$; $bn256$; $pkcs12$

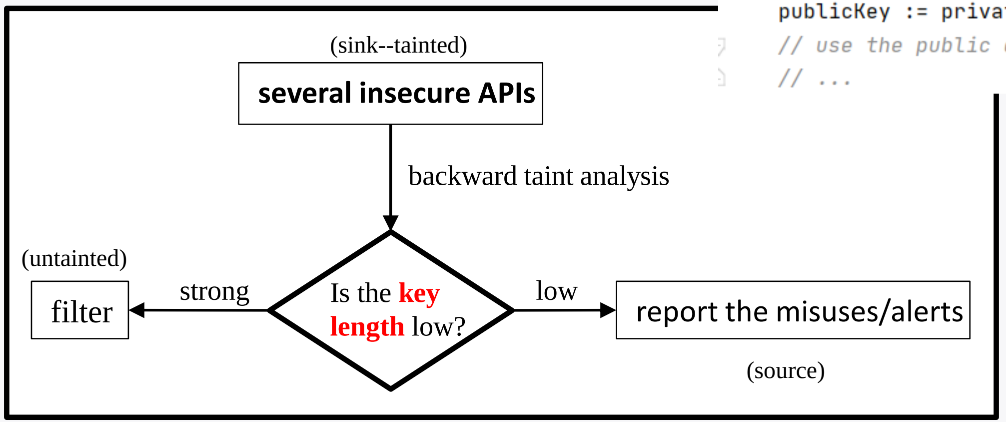
The converted IR (SSA form) $\xrightarrow{\text{which is triggered}}$ **several insecure APIs** $\rightarrow$ report the misuses/alerts

Design

02 Key Length Tracking

Backward Taint Tracking
(sink to source)
R-01 & R-02: Cryptographic Algorithms

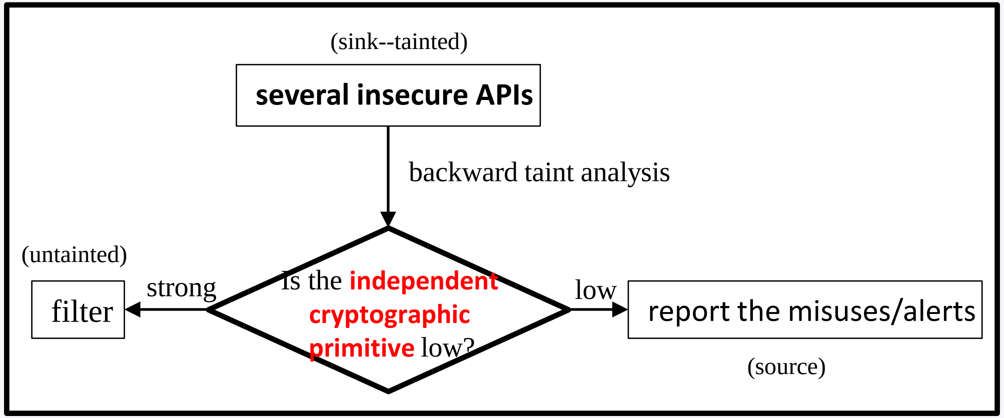
```go
func ExampleRSA() {
    bits := 1024
    // The GenerateKey method takes in a reader that returns random bits, and
    // the number of bits
    privateKey, err := rsa.GenerateKey(rand.Reader, bits)
    if err != nil : err *
    // The public key is a part of the *rsa.PrivateKey struct
    publicKey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



Design

- 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).

$$h := hmac.New(\boxed{md5.New}, []byte\ (key))$$



03 Function Nesting Tracking

Backward Taint Tracking

(sink to source)

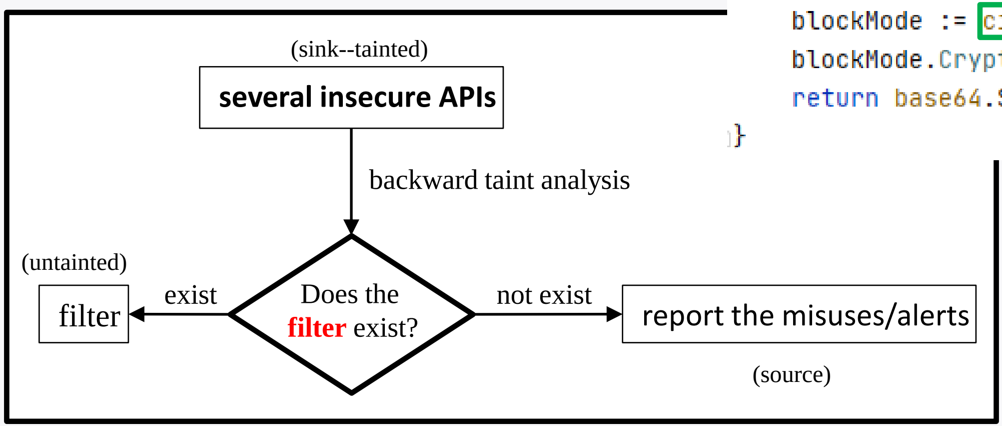
R-01 & R-02: Cryptographic Algorithms

R-08: HTTP

Design

04 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



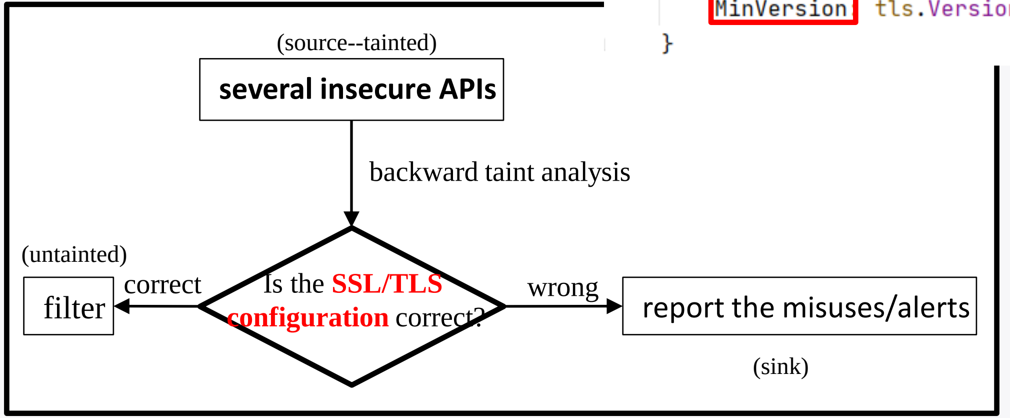

```
func AESCBCEnc(plaintext []byte) (string,error) {
    key, _ := hex.DecodeString( s: "6368616e676520746869732070617373")
    block, err := aes.NewCipher(key)
    if err != nil : err ✱
    ciphertext := make([]byte, aes.BlockSize+len(plaintext))
    iv := ciphertext[:aes.BlockSize]
    if _, err := io.ReadFull(rand.Reader, iv); err != nil : err ✱
    blockMode := cipher.NewCBCEncrypter(block, iv)
    blockMode.CryptBlocks(ciphertext,plaintext)
    return base64.StdEncoding.EncodeToString(ciphertext) , nil
}
```

filter does not exist

filter exists

Design

```
func 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,
    }
```



05 SSL/TLS Tracking

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

Talk Outline

Motivation	Rules	CryptoGo Design	Evaluation	Conclusion
Why did we start this work?	How to classify cryptographic algorithm and derive detection rules?	How does it work?	How is the performance?	Conclusions and reflections

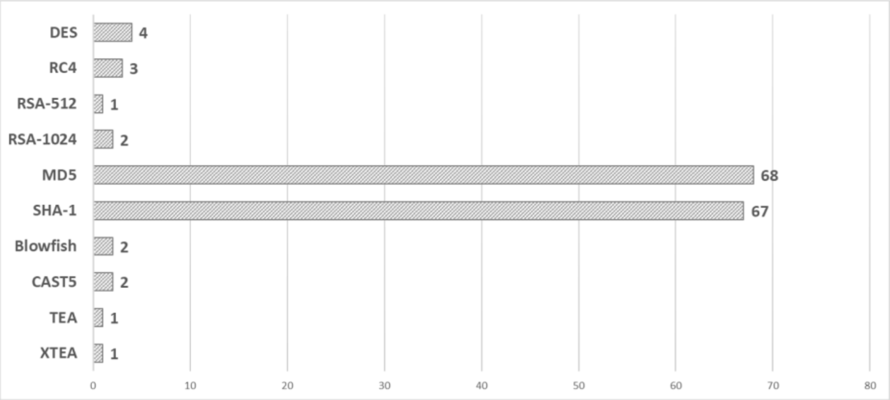
Implementation and Experimental Setup

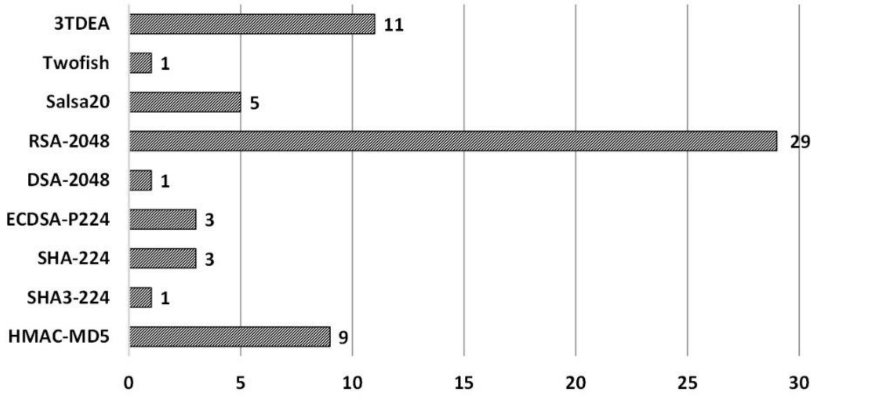
- Around 2,236 lines of Go code to realize the construction of our taint analyzers
- Dataset from GitHub:
  - Sort by the most number of stars
  - Crawled 120 open source Go projects in total.
  - The average stars and forks are 5.9k and 562.62, respectively.
  - The maximum, minimum and average Line 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.

Talk Outline

Motivation	Rules	CryptoGo Design	Evaluation	Conclusion
Why did we start this work?	How to classify cryptographic algorithm and derive detection rules?	How does it work?	How is the performance?	Conclusions and reflections

Conclusion

- 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