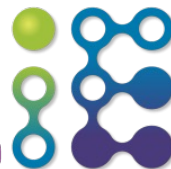
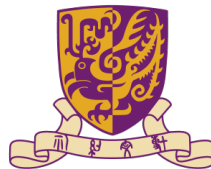


Towards Precise Reporting of Cryptographic Misuses

Yikang Chen, Yibo Liu¹, Ka Lok Wu (The Chinese University of Hong Kong), Duc V Le² (University of Bern); Sze Yiu Chau (The Chinese University of Hong Kong)

¹Now at Arizona State University

²Now at Visa Research



The Chinese University of Hong Kong
Department of Information Engineering



Cryptographic APIs



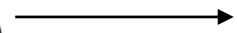
App Developers

Cryptographic APIs



App Developers

Need



Encryption



Message digests (hashes)



Digital signatures



Certificate validation



Key generation and management



Secure random number generation



Secure Socket Layer (SSL/TLS)








...

Cryptographic APIs



App Developers

Need

-  Encryption
-  Message digests (hashes)
-  Digital signatures
-  Certificate validation
-  Key generation and management
-  Secure random number generation
-  Secure Socket Layer (SSL/TLS)
- ...

Call APIs

Java Cryptography Architecture (JCA)

```
1 c = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

C libraries, e.g. OpenSSL crypto library (libcrypto)

```
1 void AES_cbc_encrypt (const unsigned char *in,\  
2                       unsigned char *out,\  
3                       size_t length,\  
4                       const AES_KEY *key,\  
5                       unsigned char *ivec,\  
6                       const int enc)
```


Cryptographic API misuses

Java Cryptography Architecture (JCA)

```
1 c = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

C libraries, e.g. OpenSSL crypto library (libcrypto)

```
1 void AES_cbc_encrypt (const unsigned char *in,\n2                       unsigned char *out,\n3                       size_t length,\n4                       const AES_KEY *key,\n5                       unsigned char *ivec,\n6                       const int enc)
```

Cryptographic API misuses

Java Cryptography Architecture (JCA)

```
1 c = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

C libraries, e.g. OpenSSL crypto library
(libcrypto)

```
1 void AES_cbc_encrypt (const unsigned char *in,\  
2                       unsigned char *out,\  
3                       size_t length,\  
4                       const AES_KEY *key,\  
5                       unsigned char *ivec,\  
6                       const int enc)
```

be misused due to

Weak cipher?

Hardcoded key, IVs or salts?

Collision-prone hashing?

Legacy SSL versions?

Broken certificate validation?

Non-CSPRNGS?

...



Cryptographic API misuse detectors

- Previous works on statically detecting potential cryptographic misuses
 - MalloDroid: SSL/TLS usage in Android applications [CCS '12]
 - CryptoLint: Cryptographic misuse detection in Android applications [CCS '13]
 - CryptoREX: Cryptographic misuse detection in IoT Devices [RAID '19]
 - CogniCryptSAST with CrySL: using domain specific languages to validate crypto misuse in Java and Android applications [TSE '19]
 - CryptoGuard: High-precision detection in Java and Android applications [CCS '19]



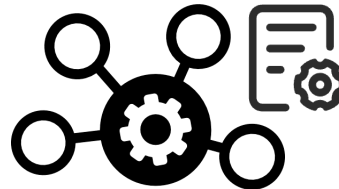
Cryptographic API misuse detectors

- Previous works on statically detecting potential cryptographic misuses
 - MalloDroid: SSL/TLS usage in Android applications [CCS '12]
 - CryptoLint: Cryptographic misuse detection in Android applications [CCS '13]
 - **CryptoREX: Cryptographic misuse detection in IoT Devices [RAID '19]**
 - **CogniCryptSAST with CrySL: using domain specific languages to validate crypto misuse in Java and Android applications [TSE '19]**
 - **CryptoGuard: High-precision detection in Java and Android applications[CCS '19]**

Open-source State-of-the-art detectors for IoT and Android apps



Applications



Static Detectors



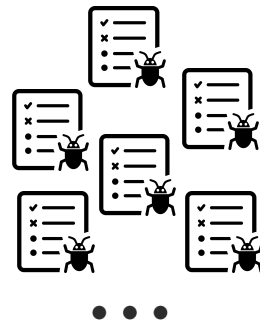
Misuse Alarms

Motivation

- Status quo
 - The benchmarks shows **high precision**.
 - The empirical results shows **plentiful violations** in real-world applications at the scale of thousands.

Motivation

- Status quo
 - The benchmarks shows **high precision**.
 - The empirical results shows **plentiful violations** in real-world applications at the scale of thousands.
- Motivation
 - If bug detectors report too many false alarms, developers will refuse to use them.
[Why don't software developers use static analysis tools to find bugs?, ICSE13]
 - Are the violations actual cryptographic misuses or false alarms?



Methodology

- We collected a dataset for evaluation

- For CryptoGuard and CogniCryptSAST:

- Collected 3489 apks from an open-source Android applications repository



- For CryptoRex:

- Collected 1177 firmware images from same vendors according to its paper

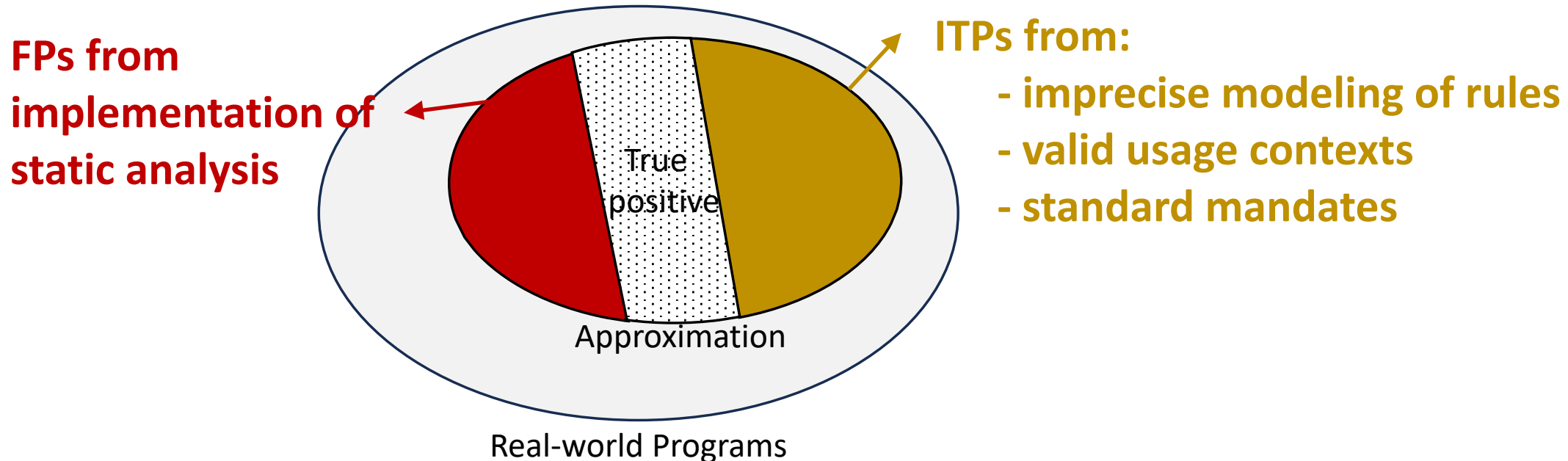
Methodology

- We collected a dataset for evaluation
 - For CryptoGuard and CogniCryptSAST:
 - Collected 3489 apks from an open-source Android applications repository
 - For CryptoRex:
 - Collected 1177 firmware images from same vendors according to its paper
- We manually analyzed the alarms using the following procedures
 - **Merge** alarms based on reported misuses' method signature.
 - **Sort** the merged alarms by their occurrence in apps.
 - Analyze **root cause** of top-10 offending methods.
 - Sample additional alarms by choice.



Overview of results

- There is a gap between misuse alarms and actual vulnerabilities that can be fixed by developers.
- We consider 2 types of false alarms: **False positives** (FPs) and **Ineffectual True Positives** (ITPs).
- We conclude 19 false alarm patterns from 4 types of root causes.



FP: non-existent data flows

- Broken def-use chains due to variable reassignment in CryptoGuard

```
1 public class testKeyStore {
2     private File keyStoreFile;
3     private String pwd;
4
5     KeyStore loadAppKeyStore() throws Exception {
6         KeyStore ks;
7         ks = KeyStore.getInstance(KeyStore.getDefaultType());
8         ...
9         ins = new java.io.FileInputStream(keyStoreFile);
10        ks.load(ins, pwd.toCharArray()); // backward-slice for 2nd parameter
11        return ks;
12    }
13
14    public void setKey(String pass) { this.pwd = pass; }
15 }
```

FP: non-existent data flows

```
1 java.security.KeyStore loadAppKeyStore() {
2   testKeyStore r0;    java.io.File $r1;
3   java.lang.String $r2;    java.security.KeyStore $r3;
4   java.io.FileInputStream $r4;    char[] $r5;
5   ...
6
7   r0 := @this: com.test.testKeyStore;
8   $r2 = staticinvoke <java.security.KeyStore: java.lang.String getDefaultType()>();
9   $r3 = staticinvoke <java.security.KeyStore: java.security.KeyStore getInstance(java.lang.String)>($r2);
10  ...
11
12  $r2 = r0.<com.test.testKeyStore: java.lang.String pwd>;
13
14  $r5 = virtualinvoke $r2.<java.lang.String: char[] toCharArray()>();
15
16  virtualinvoke $r3.<java.security.KeyStore: void load(java.io.InputStream, char[])>($r4, $r5);
17  ...
18 }
```

FP: non-existent data flows

```
1 java.security.KeyStore loadAppKeyStore() {
2   testKeyStore r0;    java.io.File $r1;
3   java.lang.String $r2;    java.security.KeyStore $r3;
4   java.io.FileInputStream $r4;    char[] $r5;
5   ...
6
7   r0 := @this: com.test.testKeyStore;
8   $r2 = staticinvoke <java.security.KeyStore: java.lang.String getDefaultType()>();
9   $r3 = staticinvoke <java.security.KeyStore: java.security.KeyStore getInstance(java.lang.String)>($r2);
10  ...
11
12  $r2 = r0.<com.test.testKeyStore: java.lang.String pwd>;
13
14  $r5 = virtualinvoke $r2.<java.lang.String: char[] toCharArray()>();
15
16  virtualinvoke $r3.<java.security.KeyStore: void load(java.io.InputStream, char[])>($r4, $r5);
17  ...
18 }
```

Use set: {\$r5}

FP: non-existent data flows

```
1 java.security.KeyStore loadAppKeyStore() {
2   testKeyStore r0;    java.io.File $r1;
3   java.lang.String $r2;    java.security.KeyStore $r3;
4   java.io.FileInputStream $r4;    char[] $r5;
5   ...
6
7   r0 := @this: com.test.testKeyStore;
8   $r2 = staticinvoke <java.security.KeyStore: java.lang.String getDefaultType()>();
9   $r3 = staticinvoke <java.security.KeyStore: java.security.KeyStore getInstance(java.lang.String)>($r2);
10  ...
11
12  $r2 = r0.<com.test.testKeyStore: java.lang.String pwd>;
13
14  $r5 = virtualinvoke $r2.<java.lang.String: char[] toCharArray()>();
15  
16  virtualinvoke $r3.<java.security.KeyStore: void load(java.io.InputStream, char[])>($r4, $r5);
17  ...
18 }
```

Use set: {\$r5, \$r2}

FP: non-existent data flows

```
1 java.security.KeyStore loadAppKeyStore() {
2   testKeyStore r0;    java.io.File $r1;
3   java.lang.String $r2;    java.security.KeyStore $r3;
4   java.io.FileInputStream $r4;    char[] $r5;
5   ...
6
7   r0 := @this: com.test.testKeyStore;
8   $r2 = staticinvoke <java.security.KeyStore: java.lang.String getDefaultType()>();
9   $r3 = staticinvoke <java.security.KeyStore: java.security.KeyStore getInstance(java.lang.String)>($r2);
10  ...
11
12  $r2 = r0.<com.test.testKeyStore: java.lang.String pwd>;
13
14  $r5 = virtualinvoke $r2.<java.lang.String: char[] toCharArray()>();
15
16  virtualinvoke $r3.<java.security.KeyStore: void load(java.io.InputStream, char[])>($r4, $r5);
17  ...
18 }
```



Use set: {\$r5, \$r2, r0.pwd}

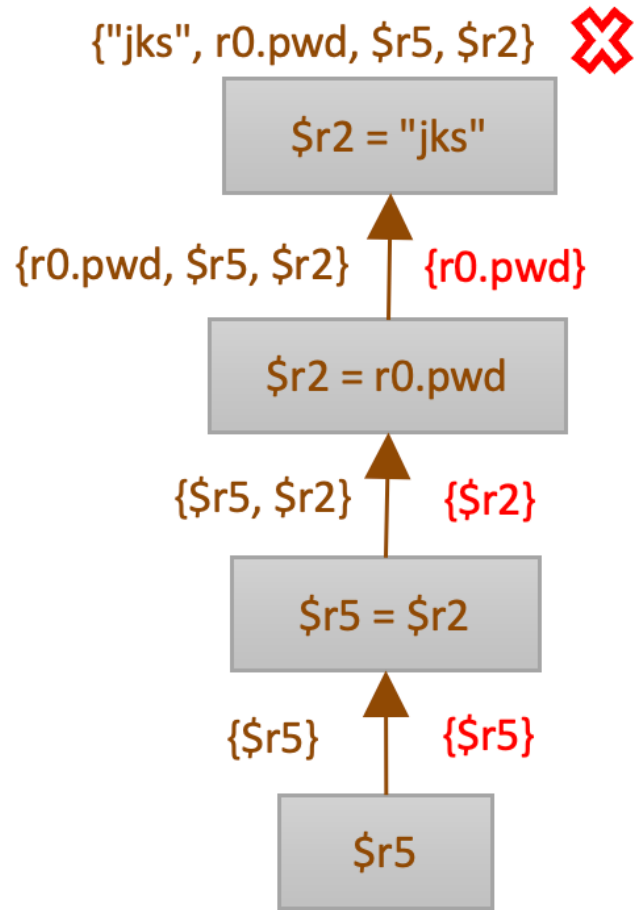
FP: non-existent data flows

```
1 java.security.KeyStore loadAppKeyStore() {
2   testKeyStore r0;    java.io.File $r1;
3   java.lang.String $r2;    java.security.KeyStore $r3;
4   java.io.FileInputStream $r4;    char[] $r5;
5   ...
6
7   r0 := @this: com.test.testKeyStore;
8   $r2 = staticinvoke <java.security.KeyStore: java.lang.String getDefaultType()>();
9   $r3 = staticinvoke <java.security.KeyStore: java.security.KeyStore getInstance(java.lang.String)>($r2);
10  ...
11
12  $r2 = r0.<com.test.testKeyStore: java.lang.String pwd>;
13
14  $r5 = virtualinvoke $r2.<java.lang.String: char[] toCharArray()>();
15
16  virtualinvoke $r3.<java.security.KeyStore: void load(java.io.InputStream, char[])>($r4, $r5);
17  ...
18 }
```

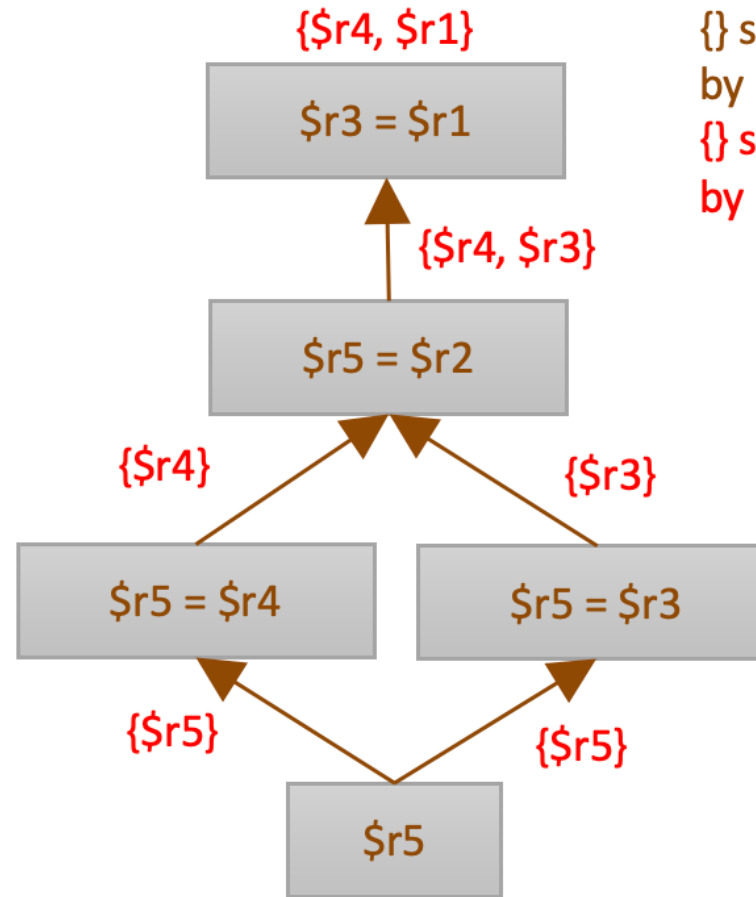
However, the method `KeyStore.getDefaultType()` returns a constant String "jks", considered as constant password

Use set: {\$r5, \$r2, r0.pwd, "jks"}

Safe refinement of def-use chains



(a)



(b)

`{}` shows used objects tracked by CryptoGuard
`{}` shows used objects tracked by refined CryptoGuard

Refined results

- We implemented the refinement and rerun it on the same data set.

Rule	Misuses	FPs (%)	Rule	Misuses	FPs (%)
1, 2 [*]	972	599 (61.63%)	10	150	20 (13.33%)
3	364	118 (32.42%)	12	490	82 (16.73%)
8	105	13 (12.38%)	13	1510	1438 (95.23%)

* CryptoGuard combines these two into one type of alarm

FP: incorrect detection of hardcoded arrays

- A bug in detection of hard-coded arrays in CogniCryptSAST

```
1 public void test() throws Exception {
2     KeyStore kS = KeyStore.getInstance(KeyStore.getDefaultType());
3     kS.load(null, getPassword());
4 }
5
6 public char[] getPassword() {
7     byte[] pass = new byte[256]; → Jimple IR
8     SecureRandom sR = new SecureRandom();
9     sR.nextBytes(pass);
10    return bytesToChars(pass);
11 }
```

```
byte[] r0;
r0 = newarray (byte)[256];
```

FP: incorrect detection of hardcoded arrays

- A bug in detection of hard-coded arrays in CogniCryptSAST

```
1 public void test() throws Exception {
2     KeyStore kS = KeyStore.getInstance(KeyStore.getDefaultType());
3     kS.load(null, getPassword());
4 }
5
6 public char[] getPassword() {
7     byte[] pass = new byte[256];
```

Jimple IR → `byte[] r0;`
`r0 = newarray (byte)[256];`



- Idiosyncrasies of the IR need to be carefully considered.
- Detectors need thorough testing and evaluation based on real-world datasets with sufficient sample size.

ITP: Overly broad blacklists

- Reasonable iteration counts considered insecure in CogniCryptSAST
 - Requires the PBE iteration count: $10000 > 1000$ (minimum requirement of the time)

ITP: Overly broad blacklists

- Reasonable iteration counts considered insecure in CogniCryptSAST
 - Requires the PBE iteration count: $10000 > 1000$ (minimum requirement of the time)
- Reasonable key sizes considered insecure in CryptoGuard
 - Requires EC key size to be 512 bit long \approx RSA 15360 \gg RSA 2048
 - ITP Examples:
 - A 256-bit key for ED25519
 - A 384-bit key for ECDSA-384
 - Android 6-13 (API 23- 33): default key size EC 256, RSA 2048

ITP: Overly broad blacklists

- Reasonable iteration counts considered insecure in CogniCryptSAST
 - Requires the PBE iteration count: $10000 > 1000$ (minimum requirement of the time)
- Reasonable key sizes considered insecure in CryptoGuard
 - Requires EC key size to be 512 bit long \approx RSA 15360 \gg RSA 2048
 - ITP Examples:
 - A 256-bit key for ED25519
 - A 384-bit key for ECDSA-384
 - Android 6-13 (API 23- 33): default key size EC 256, RSA 2048
- Constant seeds assumed to always make outputs of SecureRandom predictable in CryptoGuard
 - Android 7+ (API 24+): a given seed is always used as a supplement to randomness
 - Older versions depend on choices of providers

ITP: Overly narrow whitelists

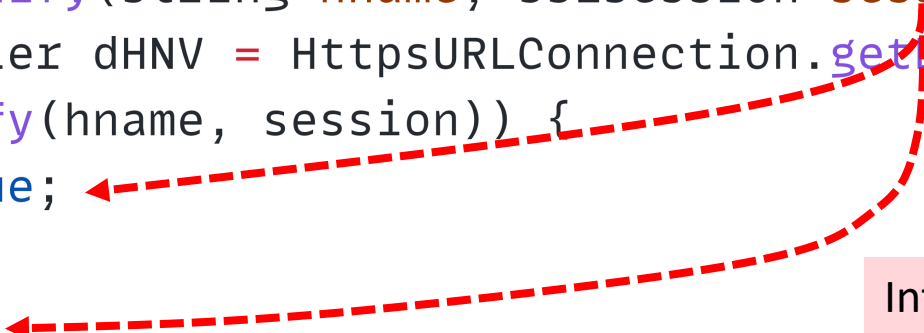
- Narrow whitelist constraints in detecting MITM in CryptoGuard
 - Requires session to influence return value of verify method via def-use relations

```
1 public class Verifier implements HostnameVerifier {
2     ...
3     @Override
4     public boolean verify(String hname, SSLSession session) {
5         HostnameVerifier dHNV = HttpsURLConnection.getDefaultHostnameVerifier();
6         if (dHNV.verify(hname, session)) {
7             return true;
8         }
9         return false;
10    }
11 }
```

ITP: Overly narrow whitelists

- Narrow whitelist constraints in detecting MITM in CryptoGuard
 - Requires session to influence return value of verify method via def-use relations

```
1 public class Verifier implements HostnameVerifier {
2     ...
3     @Override
4     public boolean verify(String hname, SSLSession session) {
5         HostnameVerifier dHNV = HttpsURLConnection.getDefaultHostnameVerifier();
6         if (dHNV.verify(hname, session)) {
7             return true;
8         }
9         return false;
10    }
11 }
```



Influence return value through **control flows** (if-else) without def-use relations

ITP: Overly narrow whitelists

- IVs must come from SecureRandom in CogniCryptSAST
 - ITP example: decryption mode
- Key materials to construct SecretKeySpec must come from existing SecretKey or Key objects in CogniCryptSAST
 - ITP examples:
 - Loading the bytes from key files, network
 - Generating a byte array with SecureRandom

ITP: Overly narrow whitelists

- IVs must come from SecureRandom in CogniCryptSAST
 - ITP example: decryption mode



- Justify the adopted lower bounds by clearly citing the standards or recommendations of the time.
- **Blacklists** modeling requires to capture **sufficient conditions** of a misuse. Labels of confidence (sufficient v.s. necessary conditions) can be helpful.
- **Whitelists** modeling is hard as it requires the understanding of common **legitimate patterns**. Mining and testing code repositories can be helpful.

ITP: misuse rules v.s. usage contexts

- All usage of AES-ECB considered insecure
 - ITP example: AES-ECB can be used as the raw AES block cipher for implementing other secure modes of operation, e.g., AES-EAX mode in Google's Tink library

```
1 public final class AesEaxJce implements Aead {
2     ...
3     private static final ThreadLocal<Cipher> localEcbCipher =
4         new ThreadLocal<Cipher>() {
5             @Override
6             protected Cipher initialValue() {
7                 try {
8                     return EngineFactory.CIPHER.getInstance("AES/ECB/NOPADDING");
9                 } catch (GeneralSecurityException ex) {
10                    throw new IllegalStateException(ex);
11                }
12            }
13        };
14    ...
15 }
```

ITP: misuse rules v.s. usage contexts

- All usage of non-CSPRNGs considered insecure
 - non-CSPRNG can be used in other security-insensitive scenarios, e.g., UI animation on Android

```
1 private void calculateOut(View sceneRoot, Rect bounds, int[] outVector) {
2     sceneRoot.getLocationOnScreen(mTempLoc);
3     ...
4     double xVector = centerX - focalX;
5     double yVector = centerY - focalY;
6     if (xVector == 0 && yVector == 0) {
7         // Random direction when View is centered on focal View.
8         xVector = (Math.random() * 2) - 1;
9         yVector = (Math.random() * 2) - 1;
10    }
11    ...
12 }
```

ITP: misuse rules v.s. usage contexts

- All usage of collision-prone hash functions considered insecure
 - ITP examples: MD5 can be utilized for hashing files as an index for local cache lookup or for logging purposes, e.g. Facebook's SoLoader library

```
1 /** * Logs MD5 of lib that failed loading */
2 private String getLibHash(String libPath) {
3     String digestStr;
4     try {
5         File libFile = new File(libPath);
6         MessageDigest digest = MessageDigest.getInstance("MD5");
7         try (InputStream libInStream = new FileInputStream(libFile)) {
8             byte[] buffer = new byte[4096];
9             int bytesRead;
10            while ((bytesRead = libInStream.read(buffer)) > 0) {
11                digest.update(buffer, 0, bytesRead);
12            }
13            digestStr = String.format("%32x", new BigInteger(1, digest.digest()));
14        }
15    }
16    ...
17    return digestStr;
18 }
```

ITP: misuse rules v.s. usage contexts

- All usage of collision-prone hash functions considered insecure
 - ITP examples: MD5 can be utilized for hashing files as an index for local cache lookup or for logging purposes, e.g. Facebook's SoLoader library

```
1 /** * Logs MD5 of lib that failed loading */
2 private String getLibHash(String libPath) {
3     String digestStr;
4     try {
5         File libFile = new File(libPath);
6         MessageDigest digest = MessageDigest.getInstance("MD5");
```



- ITPs come from legitimate usage context where they provide sufficient guarantees and desirable performance.
- We recommend detectors target specific usages that are vulnerable under a well-defined threat model.

```
16     ...
17     return digestStr;
18 }
```

Generalizability of false alarm patterns

- We evaluate our found patterns on FindSecBugs.
 - An open-source industrial tool for finding security bugs in Java and Android applications based on SpotBugs.

 **Find Security Bugs**

The SpotBugs plugin for security audits of Java web applications.

Original FindSecBugs Patterns [70]	How is the pattern being modeled?	False Alarm Patterns
TrustManager that accept any certificates	report bug when there are no statements of invocation or field loading in the methods <code>checkClientTrusted()</code> , <code>checkServerTrusted()</code> or <code>getAcceptedIssuers()</code> of a class implementing <code>X509TrustManager</code> .	Pattern #10 applies
HostnameVerifier that accept any signed certificates	report bug when there are no statements of invocation or field loading in the method <code>verify()</code> of a class implementing <code>HostnameVerifier</code> .	Pattern #10 applies
SHA-1 is a weak hash function	report any <code>MessageDigest.getInstance("SHA-1");</code>	Pattern #18 applies
MD5 are weak hash functions	report any <code>MessageDigest.getInstance("MD5");</code>	Pattern #18 applies
Weak SSLContext	report any <code>SSLContext.getInstance("SSL");</code>	Pattern #13 applies
DES is insecure	report any <code>Cipher.getInstance("DES/...");</code>	Pattern #19 applies
DESede is insecure	report any <code>Cipher.getInstance("DESede/...");</code>	Pattern #19 applies
Hard-coded key	mark constant values as hardcoded in a method and report bug when methods <code>PBEKeySpec</code> or <code>SecretKeySpec</code> initialized using the marked values.	/
Static IV	report static IV when there is no <code>Cipher.getIV()</code> , no <code>SecureRandom.nextBytes()</code> , and the cipher mode is not decryption in a method initializing <code>IvParameterSpec</code> .	Avoided Pattern #11
ECB mode is insecure	report any <code>Cipher.getInstance("AES/ECB/NoPadding");</code>	Pattern #15 applies
RSA usage with short key	report any constant <code>key_size < 2048</code> in <code>KeyPairGenerator.initialize(key_size)</code>	/

Generalizability of false alarm patterns

- We evaluate our found patterns on FindSecBugs.
 - An open-source industrial tool for finding security bugs in Java and Android applications based on SpotBugs.



- 5 patterns also apply, but avoid requiring IV from SecureRandom in decryption mode.

Original FindSecBugs Patterns [70]	How is the pattern being modeled?	False Alarm Patterns
TrustManager that accept any certificates	report bug when there are no statements of invocation or field loading in the methods <code>checkClientTrusted()</code> , <code>checkServerTrusted()</code> or <code>getAcceptedIssuers()</code> of a class implementing <code>X509TrustManager</code> .	Pattern #10 applies
HostnameVerifier that accept any signed certificates	report bug when there are no statements of invocation or field loading in the method <code>verify()</code> of a class implementing <code>HostnameVerifier</code> .	Pattern #10 applies
SHA-1 is a weak hash function	report any <code>MessageDigest.getInstance("SHA-1");</code>	Pattern #18 applies
MD5 are weak hash functions	report any <code>MessageDigest.getInstance("MD5");</code>	Pattern #18 applies
Weak SSLContext	report any <code>SSLContext.getInstance("SSL");</code>	Pattern #13 applies
DES is insecure	report any <code>Cipher.getInstance("DES/...");</code>	Pattern #19 applies
DESede is insecure	report any <code>Cipher.getInstance("DESede/...");</code>	Pattern #19 applies
Hard-coded key	mark constant values as hardcoded in a method and report bug when methods <code>PBEKeySpec</code> or <code>SecretKeySpec</code> initialized using the marked values.	/
Static IV	report static IV when there is no <code>Cipher.getIV()</code> , no <code>SecureRandom.nextBytes()</code> , and the cipher mode is not decryption in a method initializing <code>IvParameterSpec</code> .	Avoided Pattern #11
ECB mode is insecure	report any <code>Cipher.getInstance("AES/ECB/NoPadding");</code>	Pattern #15 applies
RSA usage with short key	report any constant <code>key_size < 2048</code> in <code>KeyPairGenerator.initialize(key_size)</code>	/

Takeaway

- Cryptographic misuse static detectors exhibit **many unnecessary false alarms** in our real-world evaluation.

Takeaway

- Cryptographic misuse static detectors exhibit **many unnecessary false alarms** in our real-world evaluation.
- The false alarm patterns come from the **imprecision** in static analysis, **coarse modeling** of cryptographic misuses and overlooking usage **context**.

Takeaway

- Cryptographic misuse static detectors exhibit **many unnecessary false alarms** in our real-world evaluation.
- The false alarm patterns come from the **imprecision** in static analysis, **coarse modeling** of cryptographic misuses and overlooking usage **context**.
- For future works:
 - **Tighten** the decision boundary of static analysis.
 - **Thoroughly** scrutinize real-world results on top-N offending methods.
 - Use real-world patterns can help **refine** their approximations.
 - For **measuring** the security of apps, detectors must be used with **care**.

Thank You!

 Our artifact is available:

<https://github.com/kynehc/crypto-detector-evaluation-artifacts>

Questions?

 ykchen@ie.cuhk.edu.hk

 y1kang.com

Backup Slides Next

Cryptographic API misuses

- An example of hardcoded password and salt.

```
1 // misuse#1: hardcoded salt
2 byte[] salt = new byte[] {'s','a','l','t'};
3
4 // misuse#2: hardcoded password
5 char[] password = new char[] {'p','a','s','s'};
6
7 int iterationCount = 1000;
8 PBEKeySpec pbeKeySpec = new PBEKeySpec(password, salt, iterationCount, 256);
9 SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
10 SecretKey key = skf.generateSecret(pbeKeySpec);
```

Detect Cryptographic API misuses

- Detect cryptographic API misuses via static analysis

```
1 // misuse#1: hardcoded salt
2 byte[] salt = new byte[] {'s','a','l','t'};
3
4 // misuse#2: hardcoded password
5 char[] password = new char[] {'p','a','s','s'};
6
7 int iterationCount = 1000;
8
9 PBEKeySpec pbeKeySpec = new PBEKeySpec(password, salt, iterationCount, 256);
10
11 SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
12
13 SecretKey key = skf.generateSecret(pbeKeySpec);
```



e.g., Backward data flow analysis

Rules for Cryptographic API misuse

- CryptoLint, CryptoREX:

Rule 1: Don not use electronic code book (ECB) mode for encryption
Rule 2: Do not use a non-random initialization vector (IV) for ciphertext block chaining (CBC) encryption
Rule 3: Do not use constant encryption keys
Rule 4: Do not use constant salts for password-based encryption (PBE)
Rule 5: Do not use fewer than 1000 iterations for PBE
Rule 6: Do not use static seeds for random number generation (RNG) functions

Rules for Cryptographic API misuse

- CryptoGuard:

Rule 1: Do not use predictable/constant cryptographic keys.
Rule 2: Do not use predictable/constant passwords for PBE.
Rule 3: Do not use predictable/constant passwords for KeyStore.
Rule 4: Do not use custom Hostname verifiers to accept all hosts.
Rule 5: Do not use custom TrustManager to trust all certificates.
Rule 6: Do not use custom SSLSocketFactory without manual Hostname verification.
Rule 7: Do not use HTTP.
Rule 8: Do not use predictable/constant PRNG seeds.
Rule 9: Do not use cryptographically insecure PRNGs (e.g., <code>java.util.Random</code>).
Rule 10: Do not use static Salts in PBE.
Rule 11: Do not use ECB mode in symmetric ciphers.
Rule 12: Do not use static IVs in CBC mode symmetric ciphers.
Rule 13: Do not use fewer than 1,000 iterations for PBE.
Rule 14: Do not use 64-bit block ciphers (e.g., DES, IDEA, Blowfish, RC4, RC2).
Rule 15: Do not use insecure asymmetric ciphers (e.g, RSA, ECC).
Rule 16: Do not use insecure cryptographic hash (e.g., SHA1, MD5, MD4, MD2).

Rules for Cryptographic API misuse

- CogniCryptSAST with CrySL

Error Types#	Description
HardCodedError (H)	object has hardcoded value
ForbiddenMethodError (F)	object calls a forbidden method
RequiredPredicateError (R)	object has a required predicate not satisfied
TypestateError (T)	object typestate not following an expected sequence of events
ConstraintError (C)	object uses a value that is not allowed
IncompleteOperationError (I)	object does not have an expected event
NeverTypeOfError (N)	object is of a forbidden type

Overview of results

- 19 false alarm patterns from 4 types of root causes

FPs from static analysis

Pattern #1 - Broken def-use chains due to variable reassignment (CryptoGuard)

Pattern #2 - Incorrect string matching in data flow analysis (CryptoGuard)

Pattern #3 - Incorrect detection of hard-coded arrays due to a bug (CogniCrypt_{SAST})

Pattern #4 - Incorrect handling of call-return edges of CFG (CryptoREX)

Pattern #5 - Static typing information available but underutilized (CogniCrypt_{SAST})

ITPs due to modeling

Pattern #6 - Reasonable iteration counts considered insecure (CogniCrypt_{SAST})

Pattern #7 - Reasonable key sizes considered insecure (CryptoGuard)

Pattern #8 - No key-pair generator equals to insecure public key (CryptoGuard)

Pattern #9 - Constant seeds assumed to always make outputs of SecureRandom predictable (CryptoGuard)

Pattern #10 - Narrow whitelist constraints in detecting MITM issues (CryptoGuard)

Pattern #11 - All IVs must come from SecureRandom, even for decryption (CogniCrypt_{SAST})

Pattern #12 - Legitimate origins of key materials prohibited (CogniCrypt_{SAST})

Pattern #13 - Whitelist constraints ignore idiosyncrasies of Android (CogniCrypt_{SAST})

Pattern #14 - Seeding with srandom not allowed (CryptoREX)

ITPs due to usage context

Pattern #15 - All usage of AES-ECB considered insecure (CryptoGuard, CogniCrypt_{SAST}, CryptoREX)

Pattern #16 - All usages of non-CSPRNGs considered vulnerabilities (CryptoGuard, CryptoREX)

Pattern #17 - All `http://` considered vulnerabilities (CryptoGuard)

Pattern #18 - All usage of collision-prone hash functions considered vulnerabilities (CryptoGuard)

ITPs due to standard mandates

Pattern #19 - Protocol standards mandate the use of weak algorithms, modes, and constants

FP: incorrect detection of hardcoded arrays

```
1 /**
2  * Function that finds the values assigned to a soot array.
3  * @param callSite call site at which sootValue is involved
4  *
5  * @param arrayLocal soot array local variable for which values are to be found
6  * @return extracted array values
7  */
8 protected Map<String, CallSiteWithExtractedValue> extractSootArray(CallSiteWithParamIndex callSite, ExtractedValue allocSite){
```

```
9 Value arrayLocal = allocSite.getValue();
```

```
10 Body methodBody = allocSite.stmt().getMethod().getActiveBody();
11 Map<String, CallSiteWithExtractedValue> arrVal = Maps.newHashMap();
12 if (methodBody != null) {
13     Iterator<Unit> unitIterator = methodBody.getUnits().snapshotIterator();
14     while (unitIterator.hasNext()) {
15         final Unit unit = unitIterator.next();
16         if (unit instanceof AssignStmt) {
17             AssignStmt uStmt = (AssignStmt) (unit);
18             Value leftValue = uStmt.getLeftOp();
19             Value rightValue = uStmt.getRightOp();
20             if (leftValue.toString().contains(arrayLocal.toString()) && !rightValue.toString().contains("newarray")) {
21                 arrVal.put(retrieveConstantFromValue(rightValue), new CallSiteWithExtractedValue(callSite, allocSite));
22             }
23         }
24     }
25 }
26 return arrVal;
```

However, it wrongly assign “newarray (byte)[256]” to arrayLocal instead of r0.

```
28 /**
29  * Function that decides if an array is hardcoded.
30  */
31 private boolean isHardCodedArray(Map<String, CallSiteWithExtractedValue> extractSootArray) {
32     return !(extractSootArray.keySet().size() == 1 && extractSootArray.containsKey(""));
33 }
34 }
```

extractSootArray.keySet().size() is 0 ≠ 1
isHardCodedArray() returns True.

FP: incorrect handling of call-return edges

- A FP example of incorrect handling of call-return edges of CFG in CryptoREX

```
1 int __fastcall sub_5E3D0(_DWORD *a1) {
2     ...
3     *(_WORD *) (v11 + 4) = sub_5A7D4(4096);
4     ...
5 }
6 __int64 sub_464F0() {
7     struct timeval tv; // [sp+0h] [bp-1
8     gettimeofday(&tv, 0);
9     return tv.tv_usec / 1000 + 1000LL * tv.tv_sec;
10 }
11
12 int __fastcall sub_5A7D4(int a1) {
13     unsigned int v1; // register r0
14     if ( byte_ADCEC != 1 ) {
15         v1 = sub_464F0();
16         srand(v1);
17         byte_ADCEC = 1;
18     }
19     ...
20 }
```

4096 is captured by its backward slicing without considering the call-return edge from sub_464F0()