K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces

Juanru Li[1], Zhiqiang Lin[2], Juan Caballero[3], Yuanyuan Zhang[1], Dawu Gu[1]

[1] G.O.S.S.I.P, Shanghai Jiao Tong University, China

[2] The Ohio State University, USA

[3] The IMDEA Software Institute, Spain

CCS'18, Toronto, Canada
October 16, 2018

上海交通大學 Shanghai Jiao Tong University

The Ohio State University

institute imdea software

Crypto Attacks and Defenses

Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2 @ CCS 2017

Existing Researches

- Crypto misuse on Mobile platforms
  - **CryptoLint** (Android) @ *CCS 2013*
  - **iCryptoTracer** (iOS) @ *NSS 2014*
  - **NativeSpeaker** (Android) @ *Inscrypt 2017*
- Crypto algorithm identification
  - **Aligot** @ *CCS 2012*
  - **CipherXRay** @ *TDSC 2012*
  - **CryptoHunt** @ *Oakland 2017*
- Parameter extraction
  - **ReFormat** @ *ESORICS 2009*
  - **Dispatcher** @ *CCS 2009*
  - **MovieStealer** @ *Usenix Security 2013*

Crypto Keys: the Utmost Secrets

- Kerckhoffs's principle
  - *A cryptosystem should be secure even if everything about the system, **except the key**, is public knowledge*

- Attacks against crypto keys

Lest we remember @ 2009 *Heartbleed* @ 2014 *Foreshadow @ 2018*

How do we find insecure keys?

```
uint8_t Key[16];
uint8_t Data[256] = {0};

void keygen(uint8_t * key, size_t len)
{
        uint8_t seed[4];
        for ( size_t i = 0; i < 4; ++i )
                seed[i] = rand() & 0xff;
        for ( size_t i = 0; i < len; ++i )
                key[i] = seed[i % 4];
}

void encrypt( uint8_t * buf, size_t len )
{
        for ( size_t i = 0; i < len; ++i )
                buf[i] ^= Key[i % 16];
}

int main()
{
        keygen(Key, 16);
        encrypt(Data, 256);
}
```

How do we find insecure keys?

```
uint8_t Key[16];
uint8_t Data[256] = {0};

void keygen(uint8_t * key, size_t len)
{
        uint8_t seed[4];
        for ( size_t i = 0; i < 4; ++i )
                seed[i] = rand() & 0xff;
        for ( size_t i = 0; i < len; ++i )
                key[i] = seed[i % 4];
}

void encrypt( uint8_t * buf, size_t len )
{
        for ( size_t i = 0; i < len; ++i )
                buf[i] ^= Key[i % 16];
}

int main()
{
        keygen(Key, 16);
        encrypt(Data, 256);
}
```

Key with inadequate randomness

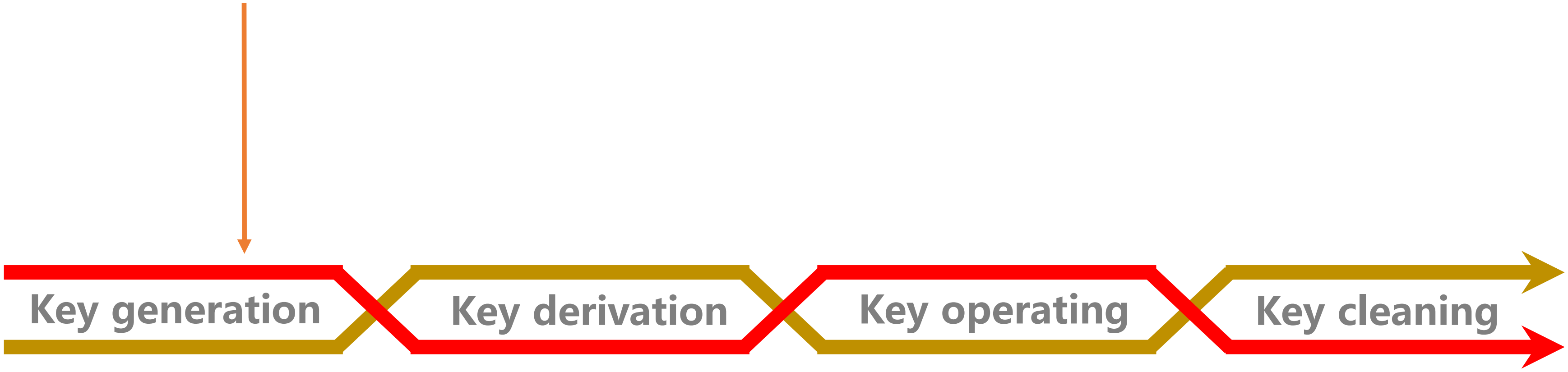
How do we find insecure keys?

```c
uint8_t Key[16];
uint8_t Data[256] = {0};

void keygen(uint8_t * key, size_t len)
{
        uint8_t seed[4];
        for ( size_t i = 0; i < 4; ++i )
                seed[i] = rand() & 0xff;
        for ( size_t i = 0; i < len; ++i )
                key[i] = seed[i % 4];
}

void encrypt( uint8_t * buf, size_t len )
{
        for ( size_t i = 0; i < len; ++i )
                buf[i] ^= Key[i % 16];
}

int main()
{
        keygen(Key, 16);
        encrypt(Data, 256);
}
```

Forget to clean the used key buffer

Cases of insecurely used crypto keys

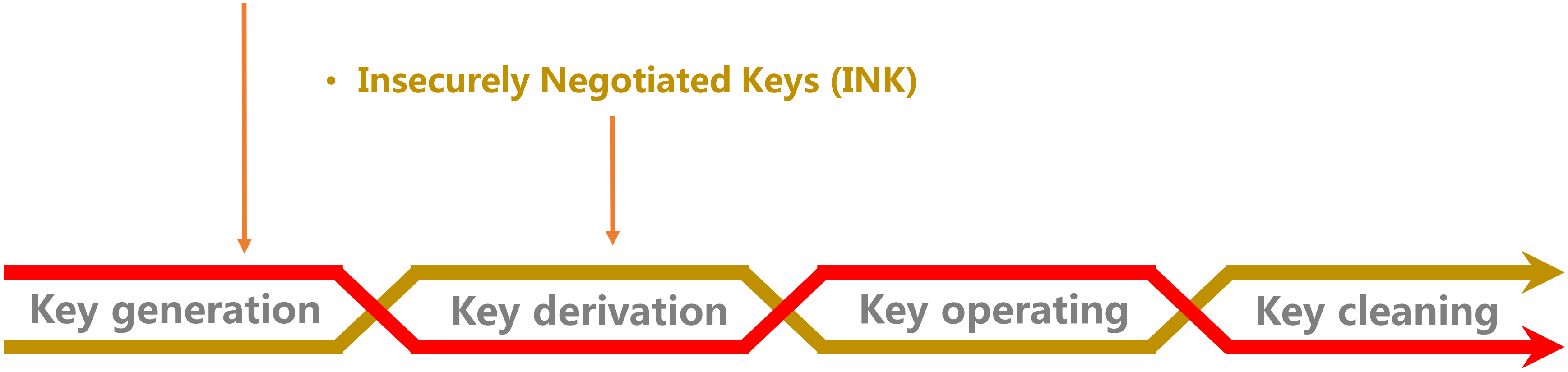
- Deterministically generated keys (DGK)



The entire lifetime of a crypto key

Cases of insecurely used crypto keys

- Deterministically generated keys (DGK)
- Insecurely Negotiated Keys (INK)



The entire lifetime of a crypto key

Cases of insecurely used crypto keys

- Deterministically generated keys (DGK)
- Insecurely Negotiated Keys (INK)
- Recoverable Keys (RK)

The entire lifetime of a crypto key

Crypto Program Analysis

```
uint8_t Key[16];
uint8_t Data[256] = {0};

void keygen(uint8_t * key, size_t len)
{
        uint8_t seed[4];
        for ( size_t i = 0; i < 4; ++i )
                seed[i] = rand() & 0xff;
        for ( size_t i = 0; i < len; ++i )
                key[i] = seed[i % 4];
}

void encrypt( uint8_t * buf, size_t len )
{
        for ( size_t i = 0; i < len; ++i )
                buf[i] ^= Key[i % 16];
}

int main()
{
        keygen(Key, 16);
        encrypt(Data, 256);
}
```

1. Locating the used ciphers

Crypto Program Analysis

2. Understanding semantics of memory buffers

```
uint8_t Key[16];
uint8_t Data[256] = {0};

void keygen(uint8_t * key, size_t len)
{
        uint8_t seed[4];
        for ( size_t i = 0; i < 4; ++i )
                seed[i] = rand() & 0xff;
        for ( size_t i = 0; i < len; ++i )
                key[i] = seed[i % 4];
}

void encrypt( uint8_t * buf, size_t len )
{
        for ( size_t i = 0; i < len; ++i )
                buf[i] ^= Key[i % 16];
}

int main()
{
        keygen(Key, 16);
        encrypt(Data, 256);
}
```

1. Locating the used ciphers

Crypto Program Analysis

```c
uint8_t Key[16];
uint8_t Data[256] = {0};

void keygen(uint8_t * key, size_t len)
{
        uint8_t seed[4];
        for ( size_t i = 0; i < 4; ++i )
                seed[i] = rand() & 0xff;
        for ( size_t i = 0; i < len; ++i )
                key[i] = seed[i % 4];
}

void encrypt( uint8_t * buf, size_t len )
{
        for ( size_t i = 0; i < len; ++i )
                buf[i] ^= Key[i % 16];
}

int main()
{
        keygen(Key, 16);
        encrypt(Data, 256);
}
```

2. Understanding semantics of memory buffers

3. Analyzing key derivation

1. Locating the used ciphers

Crypto Program Analysis

2. Understanding semantics of memory buffers

```
uint8_t Key[16];
uint8_t Data[256] = {0};

void keygen(uint8_t * key, size_t len)
{
        uint8_t seed[4];
        for ( size_t i = 0; i < 4; ++i )
                seed[i] = rand() & 0xff;
        for ( size_t i = 0; i < len; ++i )
                key[i] = seed[i % 4];
}

void encrypt( uint8_t * buf, size_t len )
{
        for ( size_t i = 0; i < len; ++i )
                buf[i] ^= Key[i % 16];
}

int main()
{
        keygen(Key, 16);
        encrypt(Data, 256);
}
```

3. Analyzing key derivation

1. Locating the used ciphers

4. Checking whether the used key is cleaned

Challenges

- **Code and algorithm diversity**
  - Proprietary ciphers
  - Customized implementations

Challenges

- Code and algorithm diversity
  - Proprietary ciphers
  - Customized implementations

- **Code complexity**
  - Large code base
  - Boundary identification of crypto functions

Challenges

- Code and algorithm diversity
  - Proprietary ciphers
  - Customized implementations

- Code complexity
  - Large code base
  - Boundary identification of crypto functions

- **Semantic recovering**
  - Deciding which memory buffers are crypto keys

Our insights

- Instead of identifying crypto algorithms (e.g., RSA)
  - **We pinpoint basic blocks related to crypto transformations directly**

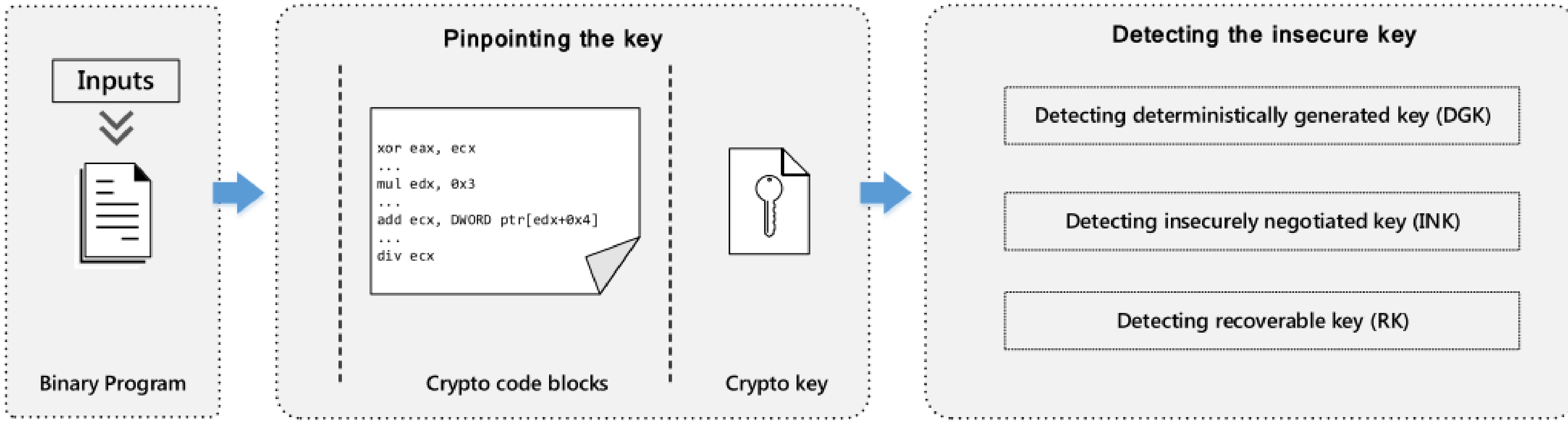
Our insights

- Instead of identifying crypto algorithms (e.g., RSA)
  - We pinpoint basic blocks related to crypto transformations directly
- Instead of analyzing program binaries
  - **We analyze execution traces to pinpoint crypto buffers**

Our insights

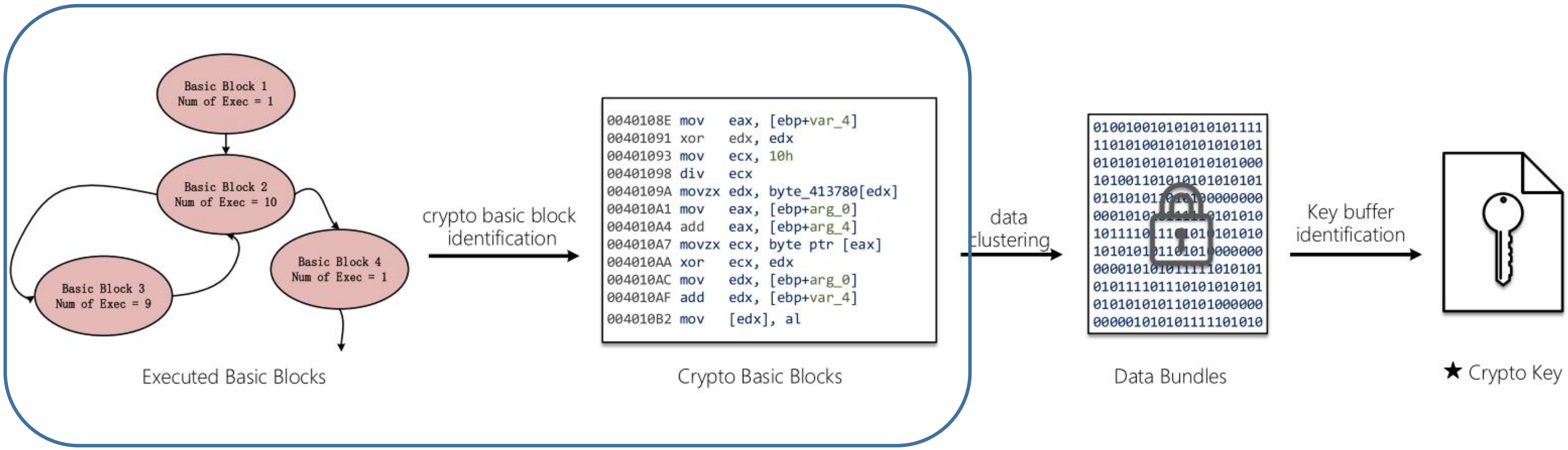
- Instead of identifying crypto algorithms (e.g., RSA)
  - We pinpoint basic blocks related to crypto transformations directly
- Instead of analyzing program binaries
  - We analyze execution traces to pinpoint crypto buffers
- Instead of statically finding specific misuses
  - **We dynamically detect insecure key**

K-Hunt



- Binary code instrumentation based on Intel's PIN framework
- Support x86/64 binary executables on Windows, Linux, and MacOS
- Comprises of two phases: **key pinpointing** and **insecure key detecting**

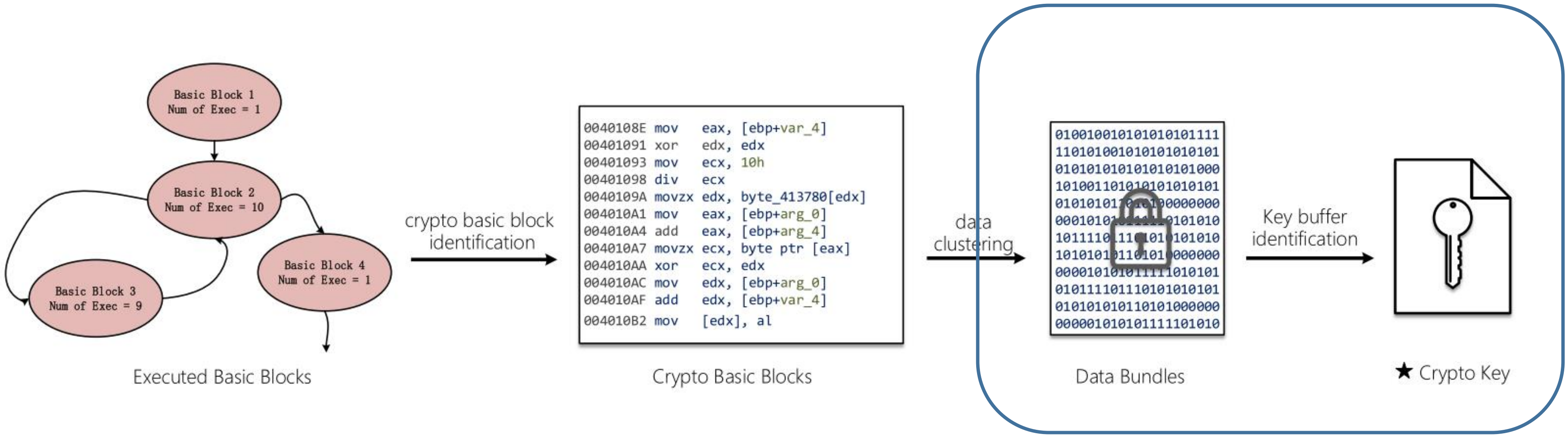
Key Pinpointing



Step-I: Crypto Basic Block Identification

- Arithmetic instructions as features
- Using multiple inputs to find data sensitive instructions
- Randomness test

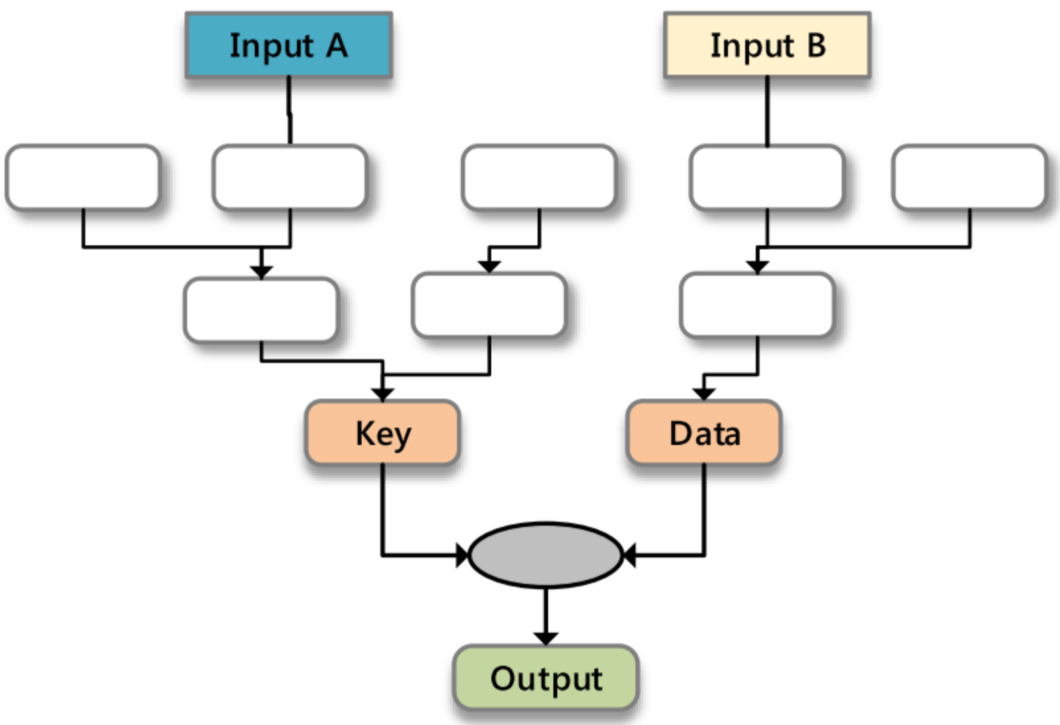
Key Pinpointing



- Step-I: Crypto Basic Block Identification
  - Arithmetic instructions as features
  - Using multiple inputs to find data sensitive instructions
  - Randomness test
- **Step-II: Crypto Key Buffer Identification**
  - Buffer size analysis
  - Execution context analysis

Insecure Key Detecting

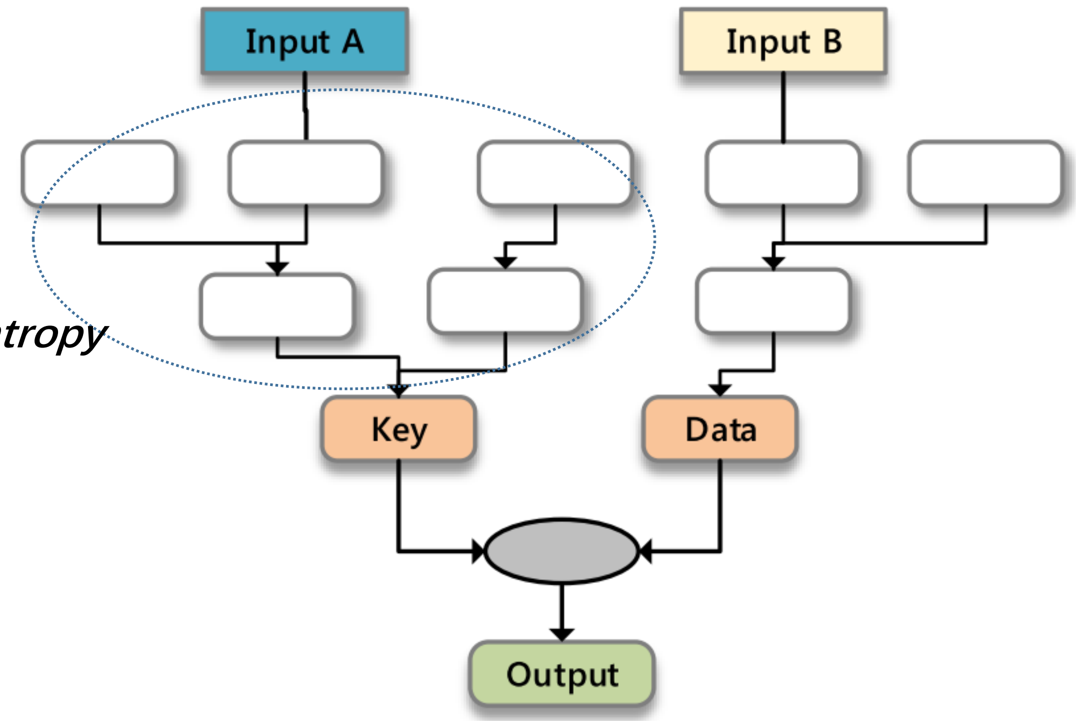
- Taint-analysis based detection



Insecure Key Detecting

- Taint-analysis based detection

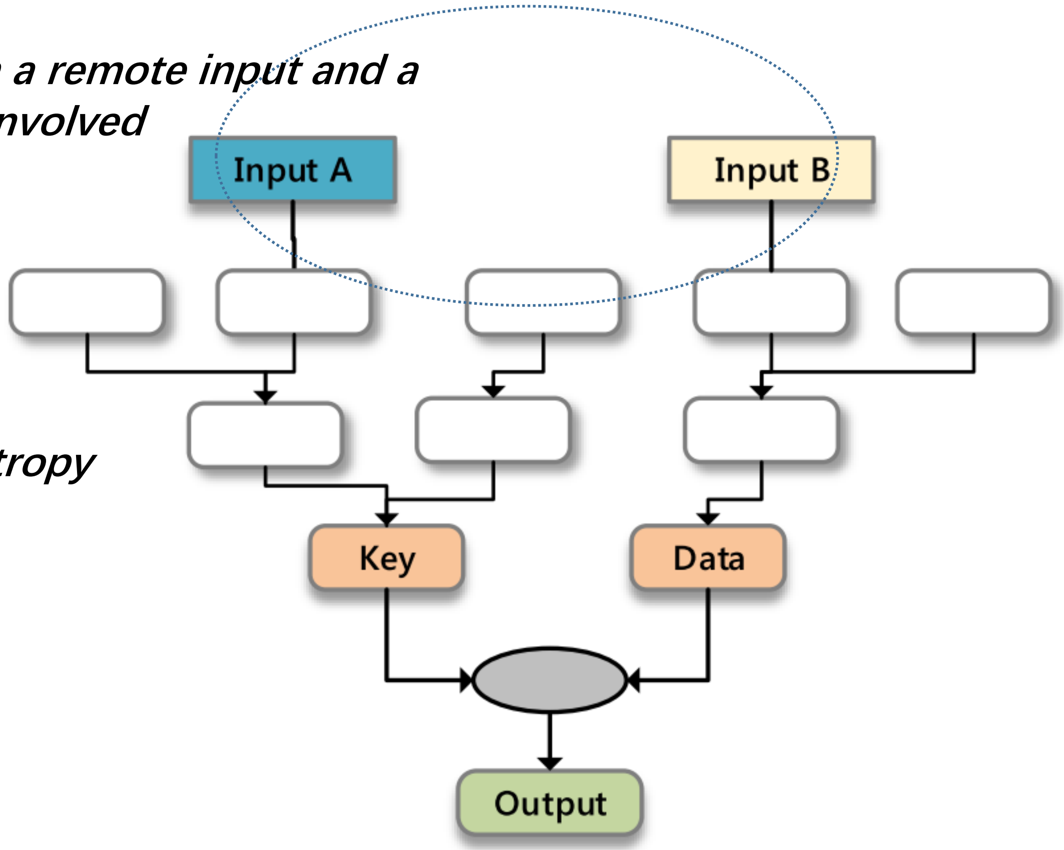
1. whether adequate entropy has been collected



Insecure Key Detecting

- Taint-analysis based detection

2. whether both a remote input and a local input are involved



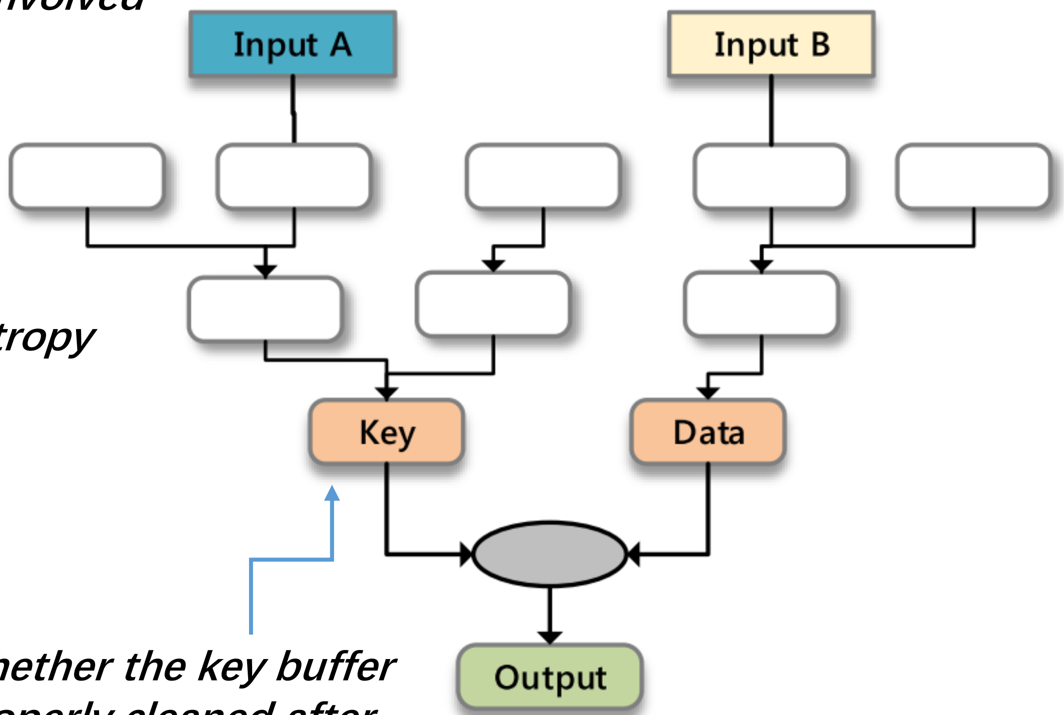
1. whether adequate entropy has been collected

Insecure Key Detecting

- Taint-analysis based detection

2. whether both a remote input and a local input are involved

1. whether adequate entropy has been collected



3. whether the key buffer is properly cleaned after the crypto operation

Experiments

- Crypto Libraries
  - 10 libraries, three ciphers (AES, RSA, ECDSA)

OpenSSL Cryptography and SSL/TLS Toolkit

Crypto++

ARM mbed

wolfSSL

libsodium

- Crypto programs
  - 15 programs with variously implemented ciphers (Including proprietary ciphers)

KeePass

7ZIP

Key Identification Results

Target	Algorithm	B1	B2	B3	N	S	IL
Botan	AES-256	53	13	7	1	240	32
	RSA-2048	1180	569	162	6	1024	256
	ECDSA	958	921	300	2	224	128
Crypto++	AES-256	1281	26	5	1	240	32
	RSA-2048	1949	924	214	6	896	256
	ECDSA	1916	1425	305	8	288	64
Libgcrypt	AES-256	126	25	3	1	240	32
	RSA-2048	565	463	153	6	896	896
	ECDSA	340	322	49	10	320	96
LibSodium	AES NI-256	7	4	4	1	240	32
	Ed25519	690	686	171	8	288	256
LibTomcrypt	AES-256	60	43	4	1	240	32
	RSA-2048	404	385	69	7	1152	1152
	ECDSA	330	274	72	4	128	97
Nettle	AES-256	38	13	3	1	240	32
	RSA-2048	411	87	61	6	1152	896
	ECDSA	186	92	39	8	288	32
mbedTLS	AES-256	44	40	13	1	240	32
	RSA-2048	154	138	39	12	1664	256
	ECDSA	255	245	47	9	384	64
OpenSSL	AES-256	58	10	4	1	240	32
	RSA-2048	210	175	41	10	1552	640
	ECDSA	188	143	17	6	192	50
WolfSSL	AES-256	50	36	4	1	240	32
	RSA-2048	295	235	36	7	1152	1152
	ECDSA	277	202	27	5	160	32

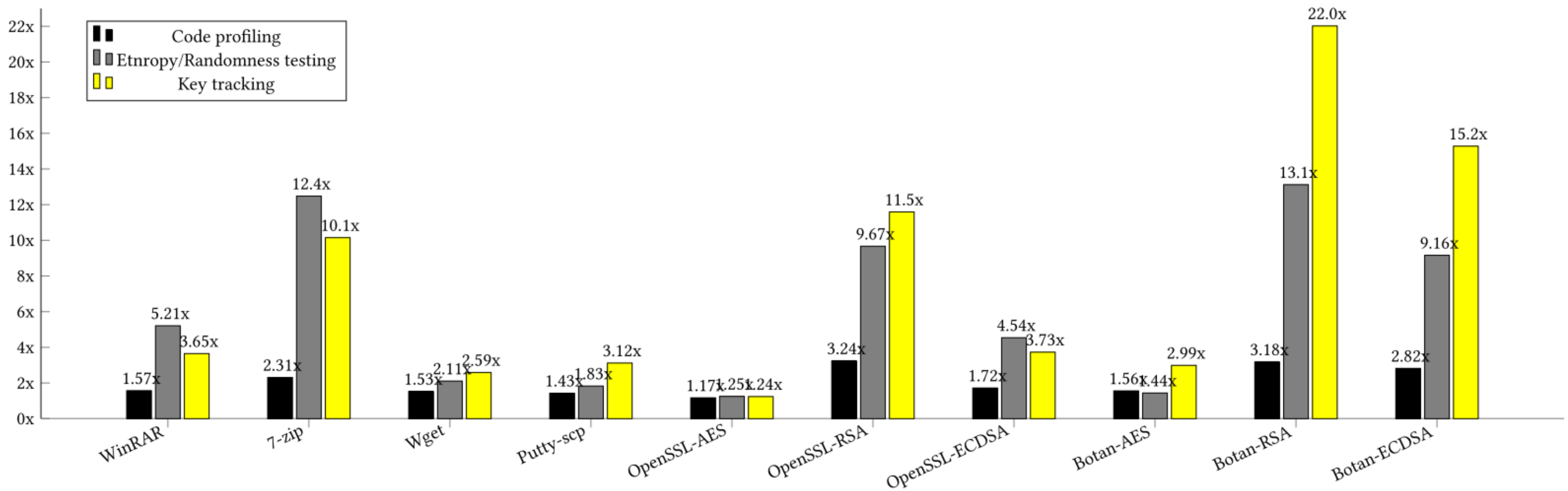
- **B1**: candidate basic blocks that contain a high arithmetic instruction ratio;
- **B2**: subset of B1 candidate basic blocks with a linear relation with the input size;
- **B3**: identified crypto basic blocks
- **N**: identified key buffers
- **S**: total size of the identified key buffers
- **IL**: input length of the identified key buffers.

Key Identification Results

Target	Algorithm	B1	B2	B3	N	S	IL
7-zip	AES NI-256	2	2	2	1	240	32
Ccrypt	AES-256	44	5	1	1	240	32
Cryptcat	Twofish	54	14	7	1	160	varied
Cryptochief	Proprietary *	23	12	1	1	8	3
Enpass	AES NI-256	8	3	3	1	240	32
Imagine	DSA-1024 *	241	72	12	5	464	928
IpMsg	AES-256	168	12	4	1	240	32
Keepass	AES-256	481	118	19	1	240	32
MuPDF	AES-128	262	46	4	1	176	16
PSCP	AES-256	195	9	5	1	240	32
Sage	ChaCha20 *	31	17	2	1	256	32
UltraSurf	RC4 *	191	79	6	1	1024	16
WannaCry	AES-128 *	26	12	3	1	352	16
Wget	AES-256	268	22	3	1	240	32
WinRAR	AES-128 *	181	58	3	1	176	32
	AES-256 *	214	51	3	1	240	48

- For 10 crypto libraries and 15 crypto programs, we successfully detected **frequently used ciphers and their key buffers**
- **Proprietary ciphers and customized implementations of standard ciphers** were detected
- **Key buffers with different layouts** are all pinpointed

Performance Overhead



Runtime overhead (times) of three pintools of K-Hunt compared to null PIN

Detected Insecurely used keys

Target	DGK	INK	RK			
			NMZ	MMZ	RKPS	RKPH
Botan	-	-	-	-	-	-
Crypto++	-	-	-	-	-	-
Libgcrypt	-	-	-	✓	-	-
LibSodium	-	-	✓	-	-	-
LibTomcrypt	-	-	✓	-	-	-
Nettle	-	-	✓	-	-	-
GnuTLS	-	-	-	✓	-	-
mbedTLS	-	-	-	✓	-	-
OpenSSL	-	-	-	✓	-	-
WolfSSL	-	-	✓	-	-	-
7-zip	-	-	-	-	✓	-
Ccrypt	-	-	-	-	-	✓
Cryptcat	-	-	-	-	-	✓
Cryptochief	✓	-	-	-	-	✓
Enpass	-	-	-	-	✓	-
Imagine	✓	-	-	-	-	-
IpMsg	-	✓	-	-	✓	-
Keepass	-	-	-	-	✓	-
MuPDF	-	-	-	-	✓	-
PSCP	-	-	-	-	-	-
Sage	-	-	-	-	✓	-
UltraSurf	-	✓	-	-	✓	-
WannaCry	-	-	-	-	✓	-
Wget	-	-	-	-	✓	-
WinRAR	-	-	-	-	-	✓

- **NMZ:** null memory zeroing
- **MMZ**: manual memory zeroing
- **RKPS**: recoverable key in program stack
- **RKPH**: recoverable key in program heap

- **22/25 tested samples are found to use insecure keys!**
- **Even well-developed crypto libraries ignore the key cleaning**
- **DGK in proprietary encryption and verification schemes**
- **INK in certificate-less network communication**

Case Study: DGK in Imagine

Imagine (an image and animation viewer) uses DSA as its registration algorithm

DSA Signing

$$r = g^k \ mod \ p \ mod \ q \qquad (1)$$

$$s = k^{-1}(H(m) + x \cdot r) \ mod \ q \qquad (2)$$

DSA Verifying

$$w = s^{-1} \ mod \ q \qquad (3)$$

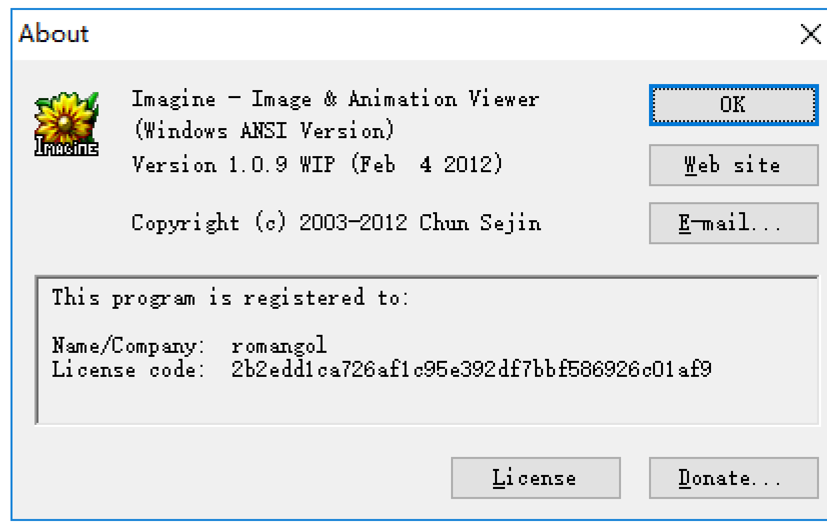
$$u_1 = H(m) \cdot w \ mod \ q \qquad (4)$$

$$u_2 = r \cdot w \ mod \ q \qquad (5)$$

$$v = (g^{u_1} \cdot y^{u_2} \ mod \ p) \ mod \ q \qquad (6)$$

"a hard-coded ***k*** leads an attacker to compute the private key ***x*** with a legal pair of signature (r, s), and thus to forge the signature"

$$x = r^{-1}(k \cdot s - H(m)) \ mod \ q$$



Case Study: RK in Libsodium

Libsodium's patch against insecurely used AES round keys:
https://github.com/jedisct1/libsodium/commit/28cac20a7bedd2ff35379874e63a33f6168ba31a

Symbolically clear the round keys after aes256gcm_(en|de)crypt()

Browse files

Fixes #617

bench-1.0.16 + stable

jedisct1 committed on 6 Nov 2017

1 parent 7b05b7d commit 28cac20a7bedd2ff35379874e63a33f6168ba31a

```
-    return crypto_aead_aes256gcm_encrypt_afternm
+    ret = crypto_aead_aes256gcm_encrypt_afternm
         (c, clen_p, m, mlen, ad, adlen, nsec, npub,
             (const crypto_aead_aes256gcm_state *) &ctx);
+    sodium_memzero(ctx, sizeof ctx);
+
+    return ret;
```

We have made responsible disclosure to the vulnerable software vendors and some of them quickly addressed the issue.

Unfortunately, some software vendors did not even response…

Conclusion

- **K-Hunt** , a dynamic analysis system to detect insecurely used keys in binary code, is developed
- Three types of insecurely used crypto keys (DGK, INK, RK) are detected using **K-Hunt**
- Insecurely used keys are found in both crypto libraries (e.g., *Libsodium*) and crypto programs (e.g., *Keepass*)

Fortune cookie

- A challenge related to the DSA case study
  - placed in **K-Hunt**'s Github repository
  - https://github.com/gossip-sjtu/k-hunt

- First 10 people to solve the challenge would receive a gift
  - Get the gift at the Ant financial desk outside

- Email the answer to loccs@sjtu.edu.cn

Thank you & Questions?

We also build new crypto libraries:

- **YogCrypt** — Chinese standard ciphers (SM2, 3, 4) in Rust
- **https://yogcrypt.org**

X

- **YogSM** — Chinese standard ciphers (SM2, 3, 4) with Intel's new hardware instructions
- **https://yogsm.org**

X