Detection of cryptographic algorithms with grap

Léonard Benedetti benedetti@mlpo.fr
Aurélien Thierry aurelien.thierry@airbus.com
Julien Francq julien.francq@airbus.com

GreHack 2017, November 17th

AIRBUS

Detection of cryptographic algorithms?

What is it?

Detect, identify and locate a cryptographic operation in a program.

What is it for?

Useful in reverse-engineering

- Time saving
- Identification of interesting areas
- Malware analysis

AIRBUS

Malware analysis: ransomware

Ransomware:

- Modern cryptography: symmetric (file encryption) + asymmetric (key management)
- Symmetric algorithms:
  - Block ciphers: AES, RC5...
  - Stream ciphers: Salsa20, ChaCha20, RC4...
- Asymmetric algorithms:
  - Key management: RSA, DH, ECDH (*e.g.* NIST curves, X25519)...

Identification of crypto algorithms within binaries:

- Automatic feature detection: "This program uses AES"
- Assist a reverser: "This function implements ChaCha20"
- Extract cryptographic material: encryption keys...

AIRBUS

Existing approaches

Constant detection and byte-level pattern matching (FindCrypt2, Signsrch, IDAScope, IDA FLIRT, YARA)

- Very quick (AES, SHA1, SHA2...)
- Easy to define patterns, hard to "get them right"
- Some algorithms don't have constants (RC4, Salsa20, ChaCha20...)
- Constant / byte modification or very light obfuscation → no detection

Function evaluation against known test values (Sybil, Aligot)

- Very precise
- Moderately difficult to write tests
- Slow
- Algorithm variant → no detection

Approach based on disassembled instructions and control flow graph (CFG)?

A quick example

ChaCha20

- Stream cipher, designed in 2008 by Daniel J. Bernstein
- Variant of Salsa20, by the same author
- Fast with a high level of security

ChaCha20

```
loc_1ABFB6:
mov     eax, [rbp+var_CC]
add     [rbp+var_DC], eax
mov     eax, [rbp+var_AC]
xor     eax, [rbp+var_DC]
rol     eax, 10h
mov     [rbp+var_AC], eax
mov     eax, [rbp+var_AC]
add     [rbp+var_BC], eax
mov     eax, [rbp+var_CC]
xor     eax, [rbp+var_BC]
rol     eax, 0Ch
mov     [rbp+var_CC], eax
mov     eax, [rbp+var_CC]
add     [rbp+var_DC], eax
mov     eax, [rbp+var_AC]
xor     eax, [rbp+var_DC]
rol     eax, 8
mov     [rbp+var_AC], eax
```

ChaCha20 encryption (LibreSSL compiled with gcc -O0)

- Repetition of ARX crypto: **add**, **xor**, **rol**

Demo: simple detection with grap

- grap "**add**->*->**xor**->**rol**" x64_libcrypto.so.37.0.0_O0
- Easy to prototype patterns
- The inferred pattern can be inspected (**-v** option)

Demo: IDA plugin

- Select the interesting areas directly in IDA
- Produce quickly usable patterns
- Apply transformations to make them generic

ChaCha20: more generic grap pattern

```
loc_1ABFB6:
mov     eax, [rbp+var_CC]
add     [rbp+var_DC], eax
mov     eax, [rbp+var_AC]
xor     eax, [rbp+var_DC]
rol     eax, 10h
mov     [rbp+var_AC], eax
mov     eax, [rbp+var_AC]
add     [rbp+var_BC], eax
mov     eax, [rbp+var_CC]
xor     eax, [rbp+var_BC]
rol     eax, 0Ch
mov     [rbp+var_CC], eax
mov     eax, [rbp+var_CC]
add     [rbp+var_DC], eax
mov     eax, [rbp+var_AC]
xor     eax, [rbp+var_DC]
rol     eax, 8
mov     [rbp+var_AC], eax
```

- Node repetition
- Conditions on opcode
- Variants: mov or lea

digraph ARX_crypto_simple {
add [**cond**="opcode is add", **repeat**=+]
mov1 [**cond**="opcode is mov or opcode is lea", **repeat**=*]
xor [**cond**="opcode is xor" **repeat**=+]
mov2 [**cond**="opcode is mov or opcode is lea", **repeat**=*]
rol [**cond**="opcode is rol" **repeat**=+]
mov3 [**cond**="opcode is mov or opcode is lea", **repeat**=*]

add -> mov1
mov1 -> xor
xor -> mov2
mov2 -> rol
rol -> mov3
}

grap overview

grap project

Patterns:

- grap "**add->*->xor->rol**" x64_libcrypto.so.37.0.0_O0
- grap pattern.grapp binary.exe
- pattern.grapp: DOT[1] file

- Standalone tool (CLI) with a Capstone-based disassembler (x86 and x86_64 only)
- IDA plugin: visually create and match patterns from IDA
- python bindings

[1] The DOT Language: `http://www.graphviz.org/content/dot-language`

AIRBUS

grap: detect graph patterns within binaries

How to quickly match subgraphs?

Control flow graphs:

- Children are ordered: `call 0x4022e0`
  - Child 1: next instruction (following address)
  - Child 2: target instruction (address: `0x4022e0`)
- Nodes have at most 2 children

→ Quick (polynomial time) algorithm for graph matching (see paper)

AIRBUS

grap: usage

https://github.com/AirbusCyber/grap

Applications:

- Malware families: detection, classification and feature extraction (REcon BRX 2017)
- Crypto detection

Build & install:

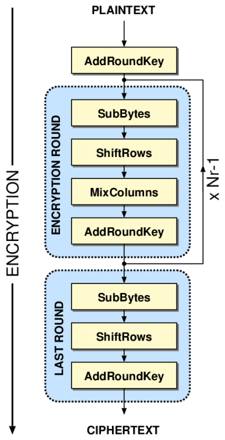
- IDA 6.95 and IDA 7.0 (32 and 64 bits) supported
- Windows: Precompiled release
- Linux: **cmake + make + sudo make install**
- Linux: tested on Ubuntu LTS (16.04) and Debian stable (9.1.0)

Designing cryptographic patterns

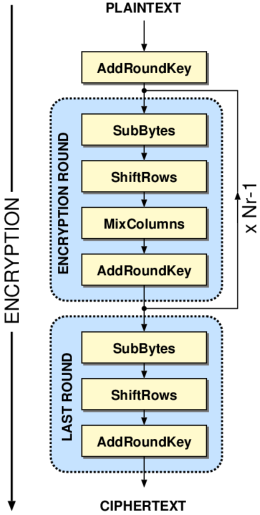
Example with AES

AES

- Block cipher, designed in 2000 by Daemen and Rijmen



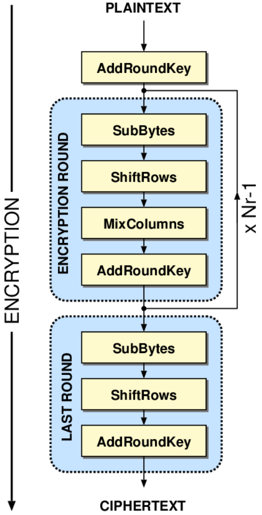
AES



Key schedule

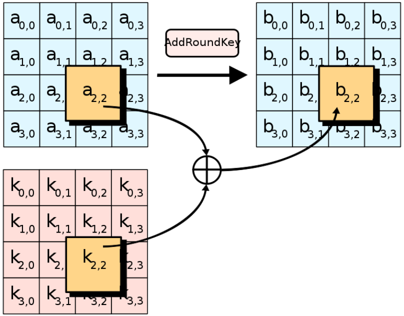
- Round keys are derived from the secret key

AES



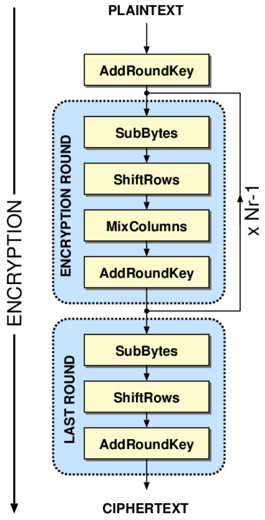
AddRoundKey

- The state is combined with the round key using XOR



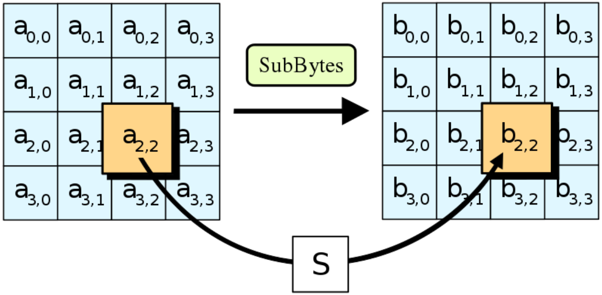
AIRBUS

AES

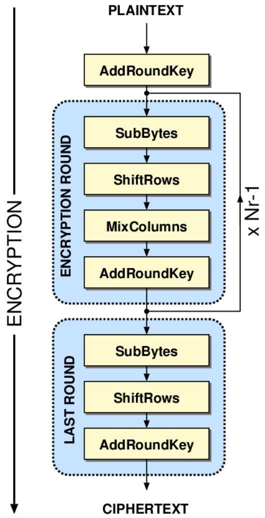


`SubBytes`

- The state is passed through a S-Box

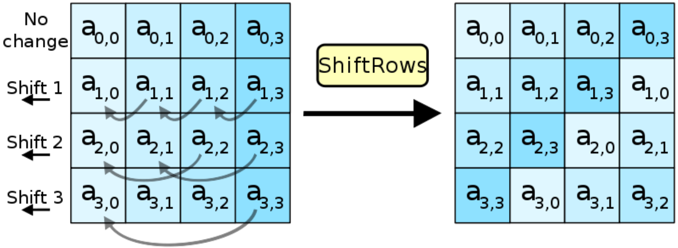


AES

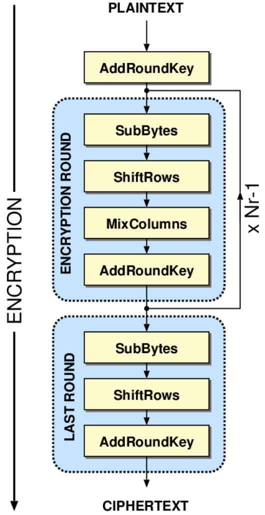


`ShiftRows`

- Cyclically shifts each row of the state

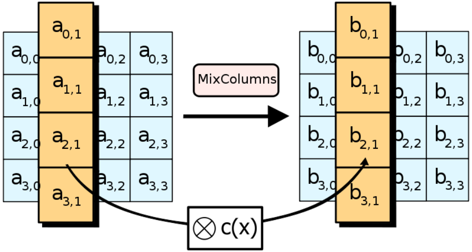


AES



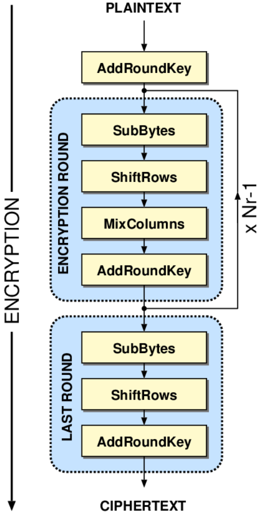
`MixColumns`

- Linear transformation in $GF(2^8)$



$$(a_3x^3 + a_2x^2 + a_1x + a_0) \times (3x^3 + x^2 + x + 2) \quad \text{mod } x^4 + 1$$

AES



- Very specific structure
- Characteristic cyclically shifts in `ShiftRows`
- Arithmetic in `MixColumns`

Design process: example with AES

1. Choosing an implementation in particular
   - LibreSSL
2. Compilation in various contexts
   - GCC, Clang
   - x86 and x64
   - Several levels of optimizations (O0, O1, O2...)

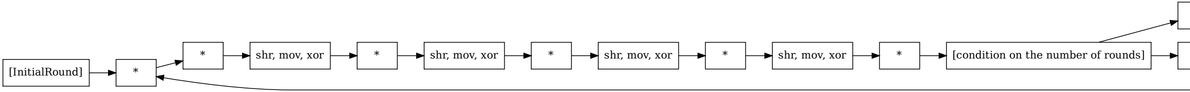
Design process: example with AES

3. Assembly code study
   - Search for invariants
   - Form of the structure
   - Analysis of semantics

4. Pattern prototyping
   - Die and retry approach
   - Attempt to generalize

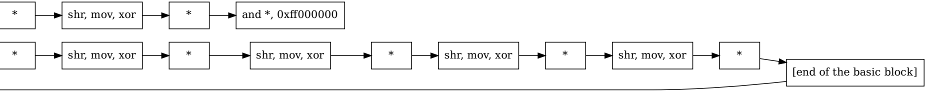
Final AES pattern

```
shr     esi, 0x18
movzx   ecx, cl
mov     esi, dword [r15+rsi*4]
mov     edi, r8d
xor     eax, dword [r12+rcx*4]
mov     rcx, r8
shr     edi, 0x18
movzx   ecx, ch
mov     edi, dword [r15+rdi*4]
add     r10, 0x20
xor     eax, dword [r13+rcx*4]
movzx   ecx, bl
```



Final AES pattern

```
and      r8d, 0xff000000
xor      r8d, r10d
xor      r9d, r8d
mov      r8d, ecx
shr      r8d, 0x10
movzx    r8d, r8b
```



Results on AES

- Effective pattern on several reference implementations
- Detection of variants (independent of the constants)
- Strongly based on the structure of the algorithm
- AES-NI detection

```
aurelien@yaourt ~/VMSharedStatic/grap-beta/samples$ grap -q /home/aurelien/grap/grap/patterns/crypto/aes_libressl_v0.1.dot libsodium.so.18.2.0
libsodium.so.18.2.0.dot - AES_NI (106)
aurelien@yaourt ~/VMSharedStatic/grap-beta/samples$ grap /home/aurelien/grap/grap/patterns/crypto/aes_libressl_v0.1.dot x64_libcrypto.so.37.0.0_00
3 unique patterns added to tree.

Test graph (x64_libcrypto.so.37.0.0_00.dot) has 318160 nodes.
2 traversal(s) possible in x64_libcrypto.so.37.0.0_00.dot: LibreSSL_AES_common (2)

LibreSSL_AES_common, match 1
LibreSSL_AES_common: 0x73dcc, shl eax, 0x10

LibreSSL_AES_common, match 2
LibreSSL_AES_common: 0x73e20, shl eax, 0x10
aurelien@yaourt ~/VMSharedStatic/grap-beta/samples$ grap /home/aurelien/grap/grap/patterns/crypto/aes_libressl_v0.1.dot x86_libcrypto.so.37.0.0_03
3 unique patterns added to tree.

Test graph (x86_libcrypto.so.37.0.0_03.dot) has 238699 nodes.
2 traversal(s) possible in x86_libcrypto.so.37.0.0_03.dot: LibreSSL_AES_common (2)

LibreSSL_AES_common, match 1
LibreSSL_AES_common: 0x45dc7, shl eax, 0x10

LibreSSL_AES_common, match 2
LibreSSL_AES_common: 0x45deb, shl eax, 0x10
aurelien@yaourt ~/VMSharedStatic/grap-beta/samples$
```

Demo

Difficulties and limitations with cryptographic patterns

- Designing effective and generic patterns is not always possible
  - Rely on semantics and topology of the CFGs, if neither is generic, the patterns won't be
  - Examples: RC4, SHA-1, SHA-2
- Cryptographic code is protean
  - Use specialized instructions: specialized opcodes (AES-NI) or vectorization (SSE, AVX, ...)
  - Ciphers can be integrated directly into other routines (mode of operation, protocols)
  - May be absent and left to the OS (*e.g.* CryptoAPI)
- Design and prototyping may take time

AIRBUS

Discussion

AIRBUS

Performance

Detect AES and ARX patterns on libsodium and LibreSSL:

```
grap -q patterns/crypto/ *
```


AIRBUS

Performance

Detect AES and ARX patterns on libsodium and LibreSSL:

```
grap -q patterns/crypto/ *
```

libsodium.so.18.2.0.grapcfg - AES_NI (106), ARX_crypto (3)
x64_libcrypto.so.41.1.0_clang_O3.grapcfg - ARX_crypto (64), LibreSSL_AES_compact (1)
x64_libcrypto.so.37.0.0_O3.grapcfg - ARX_crypto (12), LibreSSL_AES_common (1)
x64_libcrypto.so.37.0.0_O0.grapcfg - ARX_crypto (58), LibreSSL_AES_common (2)
x86_libcrypto.so.37.0.0_O0.grapcfg - ARX_crypto (58), LibreSSL_AES_common (2)

Performance

Detect AES and ARX patterns on libsodium and LibreSSL:

```
grap -q patterns/crypto/ *
```

- Overall: 25s (multithreaded)
- Disassembly: 20s
- Matching: 5s

Library	Compiler	Disassembly time	CFG size	Matching time
libsodium 1.0.12	GCC	2.1 seconds	51,866 instructions	0.6 second
LibreSSL 2.5.4 x64	Clang -O3	8.0 seconds	172,293 instructions	1.5 seconds
LibreSSL 2.3.4 x64	GCC -O3	7.2 seconds	191,307 instructions	1.6 seconds
LibreSSL 2.3.4 x64	GCC -O0	10 seconds	318,160 instructions	2.6 seconds
LibreSSL 2.3.4 x86	GCC -O0	10 seconds	346,416 instructions	2.9 seconds

Pattern detection on malware

Malware name	Symmetric crypto	Implementation	Detected	Comment
Sage	ChaCha20	custom/static	Yes	ARX
Remsec (Sauron)	RC5	custom/static	Yes	ARX
PlugX (dropper)	AES	AES-NI	Yes	
CozyDuke	AES	AES-NI	Yes	
CryptoLocker	AES	CryptoAPI	No	
Locky	AES	CryptoAPI	No	
Spora	AES	CryptoAPI	No	
WannaCry	AES	CryptoAPI	No	
NotPetya	AES+Salsa20	CryptoAPI+custom/static	No	Obfuscated
Petya	Salsa20	custom/static	No	Obfuscated

- 10 samples: 3 seconds for disassembly + matching
- ARX pattern is useful
- AES: dynamic call to CryptoAPI is predominant

AIRBUS

Detection based on control flow graphs

Complementary approach:

- Constant detection: byte level (YARA)
- **Control flow graph: implementation level**
- Function evaluation: algorithm level (Sybil)

- Implementation / CFG modification $\rightarrow$ no detection

AIRBUS

Conclusion

Conclusion

Pros

- Does not rely on constant detection
- Reliable implementation-based detection on several algorithms
- Static analysis
- Quite fast
- Easy for the analyst to quickly create and use patterns (thanks to the IDA plugin)
- Suitable for use in scripts or rules (*e.g.* for malware family identification)

Cons

- Designing generic patterns is not always possible
- Creating a generic pattern can be time consuming
- Not very effective against serious obfuscation

Conclusion

Complementary approach to crypto detection

- Functional and useful
- IDA plugin to write patterns easily
- Open source (MIT License): `https://github.com/AirbusCyber/grap`

Perspectives:

- More algorithms
- More tests on malware (quantitative analysis)
- Improve `grap` with awesome features, like “metapatterns”

AIRBUS

Thank you!

Léonard Benedetti (@mlpo_FS)
Aurélien Thierry (@yaps8)
Julien Francq

`https://github.com/AirbusCyber/grap`

AIRBUS