Recognition of binary patterns by Morphological Analysis

Aurélien Thierry (aurelien.thierry@inria.fr), Guillaume Bonfante, Joan Calvet, Jean-Yves Marion, Fabrice Sabatier

Recon 2012-06-16
Introduction

- Binary analysis

```
.text:00452C18  .text:00452C1B  sub_452C18  proc near  ; CODE XREF: sub
.text:00452C1B  .text:00452C1B  sub_452C1B  ; sub_452C1B+104
.text:00452C1B  .text:00452C1B  var_h  = dword ptr -4
.text:00452C1B  .text:00452C1B  arg_0  = dword ptr 4
.text:00452C1B  .text:00452C1B  arg_h  = dword ptr 8
.text:00452C1B  .text:00452C1B  arg_8  = dword ptr 0Ch
.text:00452C1B  push 4
.text:00452C1B  pop  eax
call  alloc
.text:00452C23  push  ebx
.text:00452C24  push  ebp
.text:00452C25  push  esi
.text:00452C26  push  edi
.mov  edi, [esp+14h+arg_0]
.text:00452C29  test  edi, edi
.text:00452C2A  jz  loc_453175
.text:00452C2B  mov  esi, [esp+14h+arg_0]
.text:00452C2C  test  esi, esi
.text:00452C2D  jz  loc_453175
.mov  eax, [esp+14h+arg_h]
.mov  ecx, 10h
cmp  eax, ecx
.jz  short loc_452C66
cmp  eax, 00h
.jz  short loc_452C62
cmp  eax, 10h
.jz  short loc_452C62
push  0FFFFFFCh
.pop  eax
.ino  loc_453178
```

Conclusion
**Introduction**

- **Binary analysis**

  ```
  .text:0021F6A  public AES_set_encrypt_key
  .text:0021F6D  AES_set_encrypt_key proc near    ; CODE XREF: AES
  .text:0021F6A  ; sub_100047B4+4
  .text:0021F6A  
  .text:0021F6A  dqword ptr [ ]
  .text:0021F6A  push 4
  .text:0021F6D  pop eax
  .text:0021F6D  call _alloca_probe
  .text:0021F72  push ebx
  .text:0021F72  push ebp
  .text:0021F7A  push esi
  .text:0021F75  push edi
  .text:0021F76  mov edi, [esp+14h+arg_0]
  .text:0021F76  test edi, edi
  .text:0021F7C  jz loc_10002401
  .text:0021F82  mov esi, [esp+14h+arg_8]
  .text:0021F82  test esi, esi
  .text:0021F86  jz loc_10002401
  .text:0021F8C  mov eax, [esp+14h+arg_9]
  .text:0021F8C  cmp eax, 88h
  .text:0021F92  jz short loc_10021F85
  .text:0021F92  cmp eax, 0C0h
  .text:0021F92  jz short loc_10021F81
  .text:0021F99  cmp eax, 108h
  .text:0021F99  jz short loc_10021F81
  .text:0021F9D  push 0FFFFFEh
  .text:0021F9D  pop eax
  .text:0021FAC  jmp loc_10002404
  ```

- **Identify libraries that do not need to be reversed**
Introduction

- Binary analysis
  
  ```assembly
  .text:10021F6A  public AES_set_encrypt_key
  .text:10021F6A  AES_set_encrypt_key proc near  ; CODE XREF: AES
  .text:10021F6A  sub_1000497B8+A
  .text:10021F6A  - dword ptr [edi]
  .text:10021F6A  push 4
  .text:10021F6A  pop  eax
  .text:10021F6A  call  _alloca_probe
  .text:10021F72  push  ebx
  .text:10021F73  push  ebp
  .text:10021F74  push  esi
  .text:10021F75  push  edi
  .text:10021F76  mov  esi, [esp+14h+arg_0]
  .text:10021F78  test edi, edi
  .text:10021F7C  jz  loc_10022A01
  .text:10021F82  mov  esi, [esp+14h+arg_0]
  .text:10021F86  test esi, esi
  .text:10021F8A  jz  loc_10022A01
  .text:10021F8E  mov  eax, [esp+14h+arg_0]
  .text:10021F92  mov  ecx, ebx
  .text:10021F97  cmp  eax, ecx
  .text:10021F99  jz  short loc_10021F85
  .text:10021F9D  cmp  eax, 0
  .text:10021F9F  jz  short loc_10021F81
  .text:10021FA0  cmp  eax, 0
  .text:10021FA2  jz  short loc_10021F81
  .text:10021FA7  cmp  eax, 0
  .text:10021FA9  jz  short loc_10021F81
  .text:10021FAB  push  8FFFFFFFH
  .text:10021FAC  jmp  loc_10021F04
  ```

- Identify libraries that do not need to be reversed

Our approach:
  - Control flow graph comparison
  - Import results in IDA
Waledac malware and OpenSSL

- Spamming botnet
- Use of cryptography for communication: RSA and AES
Waledac malware and OpenSSL

- Spamming botnet
- Use of cryptography for communication: RSA and AES

```bash
aurelien:~/R$ strings Waledac\v48\unpacked.int | grep OpenSSL
EC part of OpenSSL 0.9.8e 23 Feb 2007
ECDSA part of OpenSSL 0.9.8e 23 Feb 2007
```

- OpenSSL 0.9.8e (Feb 2007) used for cryptography
Waledac malware and OpenSSL

- Spamming botnet
- Use of cryptography for communication: RSA and AES

```
aurelien:~/R$ strings Waledac\v48\unpacked.int | grep OpenSSL
EC part of OpenSSL 0.9.8e 23 Feb 2007
ECDSA part of OpenSSL 0.9.8e 23 Feb 2007
```

- OpenSSL 0.9.8e (Feb 2007) used for cryptography
- Which functions are specifically used?
Morphological Analysis: Learning a file

Step 1: Learn
Morphological Analysis : Learning a file

Step 1 : Learn
Morphological Analysis : Learning a file

Step 1 : Learn
Morphological Analysis: Learning a file

Step 1: Learn
Morphological Analysis : Scanning a file

Step 2 : Scan
Morphological Analysis: Scanning a file

Step 2: Scan
Morphological Analysis : Scanning a file

Step 2 : Scan
Step 2: Scan
Step 2: Scan

- Gather
- Extract
- Consult
- Compare
- MA-Engine
  - Signature Detection
- MA database
- Control Flow Graph
  - Abstraction
  - CALL
  - RET
  - JCC
- USB Key
  - Control Flow Graph
  - Abstraction
  - CALL
  - RET
  - JCC
- Extract
- Gather
Step 2: Scan

Gather

Control Flow Graph
Abstraction
CALL
RET
JCC

Match
Compare
Consult
MA-Engine
Signature Detection

Extract

MA database

Consult

Compare

Output

Python
Control flow graph recovery

Control Flow Graph (CFG) : oriented graph in which nodes are instruction addresses and edges represent all paths that might be traversed during execution
Control flow graph recovery

Control Flow Graph (CFG) : oriented graph in which nodes are instruction addresses and edges represent all paths that might be traversed during execution

ASM code

cmp eax 0
jne +7
mov ecx eax
dec ecx
mul eax ecx
cmp ecx 1
jne -3
jmp +2
inc ecx
ret
Control flow graph recovery

Control Flow Graph (CFG) : oriented graph in which nodes are instruction addresses and edges represent all paths that might be traversed during execution

**ASM code**

```assembly
cmp eax 0
jne +7
mov ecx eax
dec ecx
mul eax ecx
cmp ecx 1
jne -3
jmp +2
ret
```

![Control Flow Graph Diagram](image_url)
Control Flow Graph (CFG): oriented graph in which nodes are instruction addresses and edges represent all paths that might be traversed during execution.

**ASM code**

```
cmp eax 0
jne +7
mov ecx eax
dec ecx
mul eax ecx
cmp ecx 1
jne -3
jmp +2
ret
```
Control flow graph recovery

Extraction of the control flow graph from a binary:
- Static analysis from entrypoints when possible (BeaEngine)
- Dynamic analysis otherwise (Intel’s Pintools)
Control flow graph recovery

Extraction of the control flow graph from a binary:

- Static analysis from entrypoints when possible (BeaEngine)
- Dynamic analysis otherwise (Intel’s Pintools)

Nodes of the control flow graph:

- Sequential instructions do not modify the control flow
- 4 types of instructions have an impact on the CFG (*jmp*, *call*, *jcc*, et *ret*)
Control flow graph construction & reduction

<table>
<thead>
<tr>
<th>N\textsuperscript{th} instruction</th>
<th>Control flow graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential instruction</td>
<td><img src="image1" alt="Sequential instruction diagram" /></td>
</tr>
<tr>
<td>jmp K</td>
<td><img src="image2" alt="JMP diagram" /></td>
</tr>
<tr>
<td>call K</td>
<td><img src="image3" alt="CALL diagram" /></td>
</tr>
<tr>
<td>jcc K</td>
<td><img src="image4" alt="JCC diagram" /></td>
</tr>
<tr>
<td>ret</td>
<td><img src="image5" alt="RET diagram" /></td>
</tr>
</tbody>
</table>
Control flow graph construction & reduction

<table>
<thead>
<tr>
<th>$N_{th}$ instruction</th>
<th>Control flow graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential instruction</td>
<td><img src="image1" alt="Sequential Instruction Diagram" /></td>
</tr>
<tr>
<td>jmp K</td>
<td><img src="image2" alt="Jmp K Diagram" /></td>
</tr>
<tr>
<td>call K</td>
<td><img src="image3" alt="Call K Diagram" /></td>
</tr>
<tr>
<td>jcc K</td>
<td><img src="image4" alt="Jcc K Diagram" /></td>
</tr>
<tr>
<td>ret</td>
<td><img src="image5" alt="Ret Diagram" /></td>
</tr>
</tbody>
</table>

Remove sequential instructions:

- ![Remove Sequential Instructions Diagram](image6)

Realign jumps:

- ![Realign Jumps Diagram](image7)

Remove false calls:

- ![Remove False Calls Diagram](image8)
Reduction of the control flow graph

The CFG is reducted:

- Reduce the size of the graph (fastens the algorithms)
- More abstract form: detect slight changes (junk code insertion, code re-ordering)
Reduction of the control flow graph

The CFG is reducted:
- Reduce the size of the graph (fastens the algorithms)
- More abstract form: detect slight changes (junk code insertion, code re-ordering)

Waledac with static analysis:
- 38236 nodes before reduction
- 14626 nodes after reduction
Reduction on Waledac

Figure: Part of Waledac without reduction (80 nodes) and with reduction (23 nodes)
Graph matching

Graph isomorphism detection
Graph matching

Graph isomorphism detection
Graph isomorphism detection
Graph matching

Graph isomorphism detection

![Graph isomorphism detection](image-url)
Graph matching

Graph isomorphism detection

Graph 1:
- Nodes: a, b, c, d, e
- Edges: a → b, b → a, a → c, c → a, d → e, e → d

Graph 2:
- Nodes: 0, 1, 2, 3, 4
- Edges: 0 → 1, 1 → 2, 2 → 3, 3 → 2, 4 → 1, 1 → 4

Matching:
- a ↔ 0
- b ↔ 1
- c ↔ 2
- d ↔ 3
- e ↔ 4
Graph matching

Graph isomorphism detection

Graph 1:

- Nodes: a, b, c, d, e
- Edges: a→b, b→c, c→d, d→e, e→a

Graph 2:

- Nodes: 0, 1, 2, 3, 4
- Edges: 0→1, 1→4, 4→3, 3→2, 2→0
Graph matching

Graph isomorphism detection
Graph matching: Waledac and OpenSSL

- Entire graphs are not isomorphic
Graph matching: Waledac and OpenSSL

- Entire graphs are not isomorphic
- But some parts (subgraphs) are
Graph matching: Waledac and OpenSSL

Waledac

OpenSSL
Subgraphs

- Both graphs are cut into many small subgraphs
- Generated through BFS (Breadth First Search) from each node
Subgraphs

- Both graphs are cut into many small subgraphs
- Generated through BFS (Breadth First Search) from each node
- Their size is limited (typically 24 nodes)
- Search graph isomorphismisms between subgraphs of both binaries
More on subgraphs

- From one CFG, many subgraphs are generated
- Every reachable node is in many subgraphs

Example on Waledac: 24 nodes to 8 subgraphs of size 5
Graph isomorphism problem

- Graph isomorphism has no solution in polynomial time in the general case
- The problem is in NP
- General solutions are slow
Graph isomorphism problem

- Graph isomorphism has no solution in polynomial time in the general case
- The problem is in NP
- General solutions are slow

Property (Simplification)

Our subgraphs:

- Have a root node (from which every other node is reachable)
- Each node has at most 2 children (call or jcc)
- Children are ordered

- This problem is in P
Graph isomorphism problem

- Does not exactly resolve the graph isomorphism problem
- But there are fast solutions (polynomial time)

(v) Original graph

(w) Undetected graph
Morphological analysis engine

- Signatures are subgraphs from reduced control flow graphs
- Obtained statically or dynamically
- A database (tree automata) is filled with the signatures
Morphological analysis engine

- Signatures are subgraphs from reducted control flow graphs
- Obtained statically or dynamically
- A database (tree automata) is filled with the signatures
- Learning and scanning is fast (Intel Core i5 CPU M560 2.67GHz)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Files</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learn</td>
<td>44 binaries (&lt; 2000 nodes)</td>
<td>1.2s</td>
</tr>
<tr>
<td>Scan</td>
<td>44 binaries (&lt; 2000 nodes)</td>
<td>1.1s</td>
</tr>
<tr>
<td>Learn</td>
<td>OpenSSL (28313 nodes)</td>
<td>12s</td>
</tr>
<tr>
<td>Scan</td>
<td>Waledac (14626 nodes)</td>
<td>2.0s</td>
</tr>
</tbody>
</table>
Compare Waledac and OpenSSL

- Waledac uses OpenSSL 0.9.8e (Feb 2007)
- OpenSSL learnt with reduction
Compare Waledac and OpenSSL

- Waledac uses OpenSSL 0.9.8e (Feb 2007)
- OpenSSL learnt with reduction
- One DLL is matched (libeay.dll)

<table>
<thead>
<tr>
<th>OpenSSL version</th>
<th>Comment</th>
<th>Results (common subgraphs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.8x</td>
<td>Released in May 2012</td>
<td>53</td>
</tr>
<tr>
<td>0.9.8e</td>
<td>Compiled for performance (/0x /02)</td>
<td>53</td>
</tr>
<tr>
<td>0.9.8e</td>
<td>Compiled for file size (/01)</td>
<td>1264</td>
</tr>
</tbody>
</table>
Compare Waledac and OpenSSL

- Compile OpenSSL 0.9.8e with option /O1 (size optimization)
- 1264 common subgraphs between one of the DLLs (libeay.dll) and Waledac !!

- We want to know which functions are matched
- We will compare the matched code of OpenSSL and Waledac
The larger the matched subgraphs are, the more accurate the matching.
The larger the matched subgraphs are, the more accurate the matching.

- Learns and scans with increasing number of nodes from 24
- Associate nodes that are in the largest subgraphs
The larger the matched subgraphs are, the more accurate the matching.

- Learns and scans with increasing number of nodes from 24
- Associate nodes that are in the largest subgraphs
- Outputs matched nodes for each size for IDA
Code and nodes

Waledac

OpenSSL
• Greatest subgraph found has 18 nodes
• Corresponding nodes in matched subgraphs are associated
• Then associate free nodes on matching subgraphs of lesser size
IDA plugin

With both binaries opened in IDA

- Imports the list of matched nodes
- Marks them in IDA
- Provides browsing through corresponding nodes in both instances
## Waledac / OpenSSL: common subroutines

<table>
<thead>
<tr>
<th>libeay32-098e.dll</th>
<th>subroutine</th>
<th>Waledac48.int</th>
<th>subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>10002CDE</td>
<td>CRYPTO_new_ex_data</td>
<td>00455B5C</td>
<td>sub_455B55</td>
</tr>
<tr>
<td>10002CE0</td>
<td>CRYPTO_new_ex_data</td>
<td>00455B5E</td>
<td>sub_455B55</td>
</tr>
<tr>
<td>10002D0A</td>
<td>CRYPTO_free_ex_data</td>
<td>00455B72</td>
<td>sub_455B6B</td>
</tr>
<tr>
<td>10002D0C</td>
<td>CRYPTO_free_ex_data</td>
<td>00455B74</td>
<td>sub_455B6B</td>
</tr>
<tr>
<td>10021F6D</td>
<td>AES_set_encrypt_key</td>
<td>00452C1E</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>10021F7C</td>
<td>AES_set_encrypt_key</td>
<td>00452C2D</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>10021F88</td>
<td>AES_set_encrypt_key</td>
<td>00452C39</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>10021F99</td>
<td>AES_set_encrypt_key</td>
<td>00452C4A</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>10021FA0</td>
<td>AES_set_encrypt_key</td>
<td>00452C51</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>10021FA7</td>
<td>AES_set_encrypt_key</td>
<td>00452C58</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>10021FB3</td>
<td>AES_set_encrypt_key</td>
<td>00452C64</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>10021FD9</td>
<td>AES_set_encrypt_key</td>
<td>00452C8A</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>10021FEF</td>
<td>AES_set_encrypt_key</td>
<td>00452CA0</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>100224D9</td>
<td>AES_set_encrypt_key</td>
<td>0045317D</td>
<td>sub_452C1B</td>
</tr>
<tr>
<td>100224E0</td>
<td>AES_set_decrypt_key</td>
<td>00453184</td>
<td>sub_45317E</td>
</tr>
<tr>
<td>100224F0</td>
<td>AES_set_decrypt_key</td>
<td>00453194</td>
<td>sub_45317E</td>
</tr>
<tr>
<td>100224FA</td>
<td>AES_set_decrypt_key</td>
<td>0045319E</td>
<td>sub_45317E</td>
</tr>
</tbody>
</table>

**Figure:** Matching nodes are in corresponding subroutines
Waledac / OpenSSL: common subroutines

- AES: AES_set_encrypt_key, AES_set_decrypt_key
- X509: X509_PUBKEY_set, X509_PUBKEY_get
- RSA / DSA: RSA_free, DSA_size, DSA_new_method
- BN (Big Number lib): BN_is_prime_fasttest_ex, BN_ctx_new, BN_mod_inverse
- CRYPTO: CRYPTO_lock, CRYPTO_malloc
- Misc OpenSSL routines: UI, encoding...
- ...
Comparing matched code: AES_set_encrypt_key

**Figure:** Matched code between OpenSSL (left) and Waledac (right)
Comparing code: AES_encrypt

- Not detected

**Figure:** AES_encrypt subroutine
Comparing code: AES_encrypt

- Not detected
- Control flow graph too small

**Figure:** AES_encrypt subroutine

**Figure:** Simplified AES_encrypt CFG
Waledac / OpenSSL : Findings

- OpenSSL 0.9.8e compiled for being small (option /01)
Waledac / OpenSSL : Findings

- OpenSSL 0.9.8e compiled for being small (option /01)
- Use of AES for symmetric encryption
- X.509 (certificate) handling, use of RSA and/or DSA algorithm
- Calls to primality tests (consistent with asymmetric encryption like RSA but not exclusively)
OpenSSL 0.9.8e compiled for being small (option /01)

- Use of AES for symmetric encryption
- X.509 (certificate) handling, use of RSA and/or DSA algorithm
- Calls to primality tests (consistent with asymmetric encryption like RSA but not exclusively)

- Waledac actually uses X509/RSA and AES encryption
- We were able to find out without actually reversing its code
Duqu and Stuxnet

• Static analysis on their decrypted (and unpacked) main DLLs (maindll.dll for Stuxnet and netp191.pnf for Duqu)
Duqu and Stuxnet

- Static analysis on their decrypted (and unpacked) main DLLs (maindll.dll for Stuxnet and netp191.pnf for Duqu)

First analysis:
- 26.5% of Duqu’s subgraphs are common with Stuxnet (846 subgraphs)
- 60.3% of Duqu’s nodes are in subgraphs matching with Stuxnet (2215 nodes)
- Duqu and Stuxnet are strongly related
### Duqu / Stuxnet: common subroutines

<table>
<thead>
<tr>
<th>maindll.decrypted.c</th>
<th>subroutine</th>
<th>netp191_Decrypter</th>
<th>subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>10042DB0</td>
<td>sub_10042CD2</td>
<td>100136DB</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10042DC0</td>
<td>sub_10042CD2</td>
<td>100136EB</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10042DC5</td>
<td>sub_10042CD2</td>
<td>100136F0</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10042DD3</td>
<td>sub_10042CD2</td>
<td>100136FE</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10042DE0</td>
<td>sub_10042CD2</td>
<td>1001370B</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10043116</td>
<td>sub_100430CE</td>
<td>1001353F</td>
<td>sub_100134F7</td>
</tr>
<tr>
<td>1004311F</td>
<td>sub_100430CE</td>
<td>10013548</td>
<td>sub_100134F7</td>
</tr>
<tr>
<td>10043138</td>
<td>sub_100430CE</td>
<td>10013561</td>
<td>sub_100134F7</td>
</tr>
<tr>
<td>1004314D</td>
<td>sub_100430CE</td>
<td>10013576</td>
<td>sub_100134F7</td>
</tr>
<tr>
<td>10043155</td>
<td>sub_100430CE</td>
<td>1001357E</td>
<td>sub_100134F7</td>
</tr>
<tr>
<td>10043157</td>
<td>sub_100430CE</td>
<td>10013580</td>
<td>sub_100134F7</td>
</tr>
<tr>
<td>10043161</td>
<td>sub_100430CE</td>
<td>1001358A</td>
<td>sub_100134F7</td>
</tr>
<tr>
<td>1004317B</td>
<td>sub_100430CE</td>
<td>100135A4</td>
<td>sub_100134F7</td>
</tr>
<tr>
<td>10043183</td>
<td>sub_1004317C</td>
<td>100144E3</td>
<td>sub_100144DC</td>
</tr>
<tr>
<td>1004318D</td>
<td>sub_1004317C</td>
<td>100144ED</td>
<td>sub_100144DC</td>
</tr>
</tbody>
</table>
Duqu / Stuxnet: subroutine identification

- Some of the common subroutines come from standard libraries (libc...)
- They are documented and should not be manually reversed
Duqu / Stuxnet: subroutine identification

- Some of the common subroutines come from standard libraries (libc...)
- They are documented and should not be manually reversed
- msvcr80.dll: Microsoft Visual C++ Run-Time
- How to identify its code within Duqu / Stuxnet in IDA?
Duqu / Stuxnet: libc identification

- Learn msvcr80.dll ('libc') and scan Duqu, Stuxnet
Duqu / Stuxnet: libc identification

- Learn msvcr80.dll (‘libc’) and scan Duqu, Stuxnet

IDA plugin will:
- Mark the nodes common with msvcr80.dll
- Rename the matched subroutines
Duqu / Stuxnet: common subroutines

<table>
<thead>
<tr>
<th>maindll.decrypted</th>
<th>subroutine</th>
<th>netp191_Decrypt</th>
<th>subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>10042DB0</td>
<td>sub_10042CD2</td>
<td>100136DB</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10042DC0</td>
<td>sub_10042CD2</td>
<td>100136EB</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10042DC5</td>
<td>sub_10042CD2</td>
<td>100136F0</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10042DD3</td>
<td>sub_10042CD2</td>
<td>100136FE</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10042DE0</td>
<td>sub_10042CD2</td>
<td>1001370B</td>
<td>sub_100135FD</td>
</tr>
<tr>
<td>10043116</td>
<td>msvcr80$__beginthreadex</td>
<td>1001353F</td>
<td>msvcr80$__beginthreadex</td>
</tr>
<tr>
<td>1004311F</td>
<td>msvcr80$__beginthreadex</td>
<td>10013548</td>
<td>msvcr80$__beginthreadex</td>
</tr>
<tr>
<td>10043138</td>
<td>msvcr80$__beginthreadex</td>
<td>10013561</td>
<td>msvcr80$__beginthreadex</td>
</tr>
<tr>
<td>1004314D</td>
<td>msvcr80$__beginthreadex</td>
<td>10013576</td>
<td>msvcr80$__beginthreadex</td>
</tr>
<tr>
<td>10043155</td>
<td>msvcr80$__beginthreadex</td>
<td>1001357E</td>
<td>msvcr80$__beginthreadex</td>
</tr>
<tr>
<td>10043157</td>
<td>msvcr80$__beginthreadex</td>
<td>10013580</td>
<td>msvcr80$__beginthreadex</td>
</tr>
<tr>
<td>10043161</td>
<td>msvcr80$__beginthreadex</td>
<td>1001358A</td>
<td>msvcr80$__beginthreadex</td>
</tr>
<tr>
<td>1004317B</td>
<td>msvcr80$__beginthreadex</td>
<td>100135A4</td>
<td>msvcr80$__beginthreadex</td>
</tr>
<tr>
<td>10043183</td>
<td>msvcr80$free</td>
<td>100144E3</td>
<td>msvcr80$free</td>
</tr>
<tr>
<td>1004318D</td>
<td>msvcr80$free</td>
<td>100144ED</td>
<td>msvcr80$free</td>
</tr>
</tbody>
</table>

**Figure:** Renamed subroutines matching between Duqu and Stuxnet
Highlighting msvcr80.dll in Stuxnet

Figure: Colored (yellow) code of msvcr80.dll in Stuxnet, subroutines are renamed
Duqu / Stuxnet : summary

- From the decrypted and unpacked DLLs from Stuxnet, we are able to automatically find code shared with Duqu
- Before reversing, we identify standard (msvcr80.dll) subroutines
- With IDA, we can identify and browse matching subroutines
Conclusion

- Identify used libraries
- Show code similarities
- IDA UI for browsing matched code
Conclusion

- Identify used libraries
- Show code similarities
- IDA UI for browsing matched code

Thank you

Any question? (aurelien.thierry@inria.fr)