Code Deobfuscation: Intertwining Dynamic, Static and Symbolic Approaches

Robin David & Sébastien Bardin
CEA LIST
Who are we?

#Robin David
- PhD Student at CEA LIST

#Sébastien Bardin
- Full-time researcher at CEA LIST

Where are we?

Atomic Energy Commission (CEA LIST), Paris Saclay
- Software Safety & Security Lab
  - frama C
  - BINSEC
**Context & Goal**

- Analysis of obfuscated binaries and malware (potentially self-modifying)
- Recovering high-level view of the program (e.g. CFG)
- Locating and removing obfuscation if any

**Challenges?**

- Static, dynamic and symbolic analyses are not enough used alone
- Scalability, robustness, “infeasibility queries”
Our proposal

- A new symbolic method for infeasibility-based obfuscation problems
- A combination of approaches to handle obfuscations impeding different kinds of analyses

Achievements

- A set of tools to analyse binaries (instrumentation, binary analysis and IDA integration)
- Detection of several obfuscations in packers
- Deobfuscation of the X-Tunnel malware (for which obfuscation is stripped)
Long term objectives

- dynamic disassembly
- static disassembly
- Partial safe CFG
- new input
- Execution trace
- dynamic symbolic execution
- Obfuscation information

Takeaway message

- disassembling highly obfuscated codes is challenging
- combining static, dynamic and symbolic is promising (accurate and efficient)
Agenda

1. Background
   1. Disassembling obfuscated codes
   2. Dynamic Symbolic Execution

2. Our proposal
   3. Backward-Bounded DSE
   4. Analysis combination

3. Binsec
   5. The Binsec platform

4. Case-studies
   6. Packers
   7. X-Tunnel
Disassembling obfuscated codes
Getting an exploitable representation of the program
An essential task before in-depth analysis is the CFG disassembly recovery of the program
Disassembly issues

- Code discovery
  (aka. Decoding opcodes)
- CFG reconstruction
  (aka. Building the graph, nodes & edges)
- CFG partitioning
  (aka. Finding functions, bounds etc)

  - Non-code bytes
  - Missing symbols (function addr)
  - Instruction overlapping
  - Indirect control-flow
  - Non-returning functions
  - Function code sharing
  - Non-contiguous function
  - Tail calls

*segmentation proposed in Binary Code is Not Easy, Xiaozhu Meng, Barton P. Miller
Obfuscation

Any means aiming at slowing-down the analysis process either for a human or an automated algorithm.
## Obfuscation diversity

<table>
<thead>
<tr>
<th>Control</th>
<th>Vs</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>function calls, edges</td>
<td></td>
<td>strings, constants..</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target</th>
<th>Against</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Data</td>
</tr>
<tr>
<td>CFG flattening</td>
<td>●</td>
</tr>
<tr>
<td>Jump encoding</td>
<td>●</td>
</tr>
<tr>
<td>(direct → indirect/computed)</td>
<td></td>
</tr>
<tr>
<td>Opaque predicates</td>
<td>●</td>
</tr>
<tr>
<td>VM (virtual-machines)</td>
<td>●</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>●</td>
</tr>
<tr>
<td>(self-modification, resource ciphering)</td>
<td></td>
</tr>
<tr>
<td>Call/Stack tampering</td>
<td>●</td>
</tr>
<tr>
<td>Anti-debug / anti-tampering</td>
<td>●</td>
</tr>
<tr>
<td>Signal / Exception</td>
<td>●</td>
</tr>
</tbody>
</table>

and so many others....
Opaque predicates

**Definition:** Predicate always evaluating to true (resp. false). (but for which this property is difficult to deduce)

**Taxonomy:**
- Arithmetic based
- Data-structure based
- Pointer based
- Concurrence based
- Environment based

**Corollary:**
- the dead branch allow to
  - growing the code (artificially)
  - drowning the genuine code

**eg:** $7y^2 - 1 \neq x^2$
(for any value of $x, y$ in modular arithmetic)

```
mov eax, ds:X
mov ecx, ds:Y
imul ecx, ecx
imul ecx, 7
sub ecx, 1
imul eax, eax
cmp ecx, eax
jz <trap_addr>
```
**Call stack tampering**

**Definition:** Alter the standard compilation scheme of calls and ret instructions

**Corollary:**
- real `ret` target hidden, and returnsite potentially not code
- Impede the recovery of control flow edges
- Impede the high-level function recovery

In addition, able to characterize the tampering with alignment and multiplicity.

Need to handle the tail call optimization.
Deobfuscation

- Revert the transformation (sometimes impossible)
- Simplify the code to facilitate later analyses
## Disassembly

### Notations
- **Correct**: only genuine (executable) instructions are disassembled
- **Complete**: All genuine instructions are disassembled

### Standard approaches

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic</th>
<th>symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>robust <em>(obfuscation)</em></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>correct</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>complete</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
## Disassembly

### Notations
- **Correct**: only genuine (executable) instructions are disassembled
- **Complete**: All genuine instructions are disassembled

## Standard approaches
- Static disassembly

### Table

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic</th>
<th>symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>⬤</td>
<td></td>
<td></td>
</tr>
<tr>
<td>robust (obfuscation)</td>
<td>⬤</td>
<td></td>
<td></td>
</tr>
<tr>
<td>correct</td>
<td>⬤</td>
<td></td>
<td></td>
</tr>
<tr>
<td>complete</td>
<td>⬤</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diagram
- Dynamic jump
- `jmp eax`
### Disassembly

#### Notations
- **Correct**: only genuine (executable) instructions are disassembled
- **Complete**: All genuine instructions are disassembled

#### Standard approaches
- Static disassembly
- Dynamic disassembly

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic</th>
<th>symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>⚫</td>
<td>⚫</td>
<td>⚫</td>
</tr>
<tr>
<td>robust</td>
<td>⚫</td>
<td>⚫</td>
<td>⚫</td>
</tr>
<tr>
<td>correct</td>
<td>⚫</td>
<td>⚫</td>
<td>⚫</td>
</tr>
<tr>
<td>complete</td>
<td>⚫</td>
<td>⚫</td>
<td>⚫</td>
</tr>
</tbody>
</table>

- **dynamic jump**
- **input dependent**

*Figure: Black Hat Europe 2016*
Dynamic Symbolic Execution
a.k.a Concolic Execution
Dynamic Symbolic Execution

Definition: Symbolic Execution is the mean of executing a program using symbolic values (logical symbols) rather than actual values (bitvectors) in order to obtain in-out relationship of a path.

Source Code (C)
```c
int f(int a, int b) {
    if (a < 10) {
        if (a > b) {
            printf("Ok");
        }
    }
}
```

How to reach “OK”?

Formula: \( a < 10 \land a > b \)

Solution: \( a=5, b=1 \)
Why using DSE?

More difficult to hide the semantic of the program than its syntactical form.
Intermediate Representation (IR)

→ Encode the semantic of a machine instruction

Advantages:
- bitvector size statically known
- side-effect free
- bit-precise

Shortcomings:
- no floats
- no thread modeling
- no self-modification
- no exception
- x86(32) only

Many other similar IR: REIL, BIL, VEX, LLVM IR, MIASM IR, Binary Ninja IR

---

Language DBA

<table>
<thead>
<tr>
<th>bv</th>
<th>bitvector (constant value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>l :=</td>
<td>loc (addr + offset)</td>
</tr>
<tr>
<td>e :=</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>@ [ e ] (read memory)</td>
</tr>
<tr>
<td></td>
<td>e ◇ e</td>
</tr>
<tr>
<td>lhs :=</td>
<td>v  (variable)</td>
</tr>
<tr>
<td></td>
<td>v{i,j} (extraction)</td>
</tr>
<tr>
<td></td>
<td>@[ e ] (write memory)</td>
</tr>
<tr>
<td>inst :=</td>
<td>lhs := e</td>
</tr>
<tr>
<td></td>
<td>goto e</td>
</tr>
<tr>
<td></td>
<td>ITE (c)? goto l1; goto l2</td>
</tr>
<tr>
<td></td>
<td>assert e</td>
</tr>
</tbody>
</table>
## DBA example

Decoding: \texttt{imul eax, dword ptr[esi+0x14], 7}

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>res32</td>
<td>( @[esi_{(32)} + 0x14_{(32)}] \times 7_{(32)} )</td>
</tr>
<tr>
<td>temp64</td>
<td>( \text{exts } @[esi_{(32)} + 0x14_{(32)}]<em>{64} \times \text{exts } 7</em>{(32)}_{64} )</td>
</tr>
<tr>
<td>OF</td>
<td>( \text{temp64}<em>{(64)} \neq \text{exts res32}</em>{(32)}_{64} )</td>
</tr>
<tr>
<td>SF</td>
<td>( \perp )</td>
</tr>
<tr>
<td>ZF</td>
<td>( \perp )</td>
</tr>
<tr>
<td>CF</td>
<td>( \text{OF}_{(1)} )</td>
</tr>
<tr>
<td>eax</td>
<td>( \text{res32}_{(32)} )</td>
</tr>
</tbody>
</table>
DSE on a switch

Source Code (C)
enum E = {A, B, C}
int myfun(int x) {
    switch(x) {
        case A: x+=0; break;
        case B: x+=1; break;
        case C: x+=2; break;
    }
}

x86 assembly
push ebp
mov ebp, esp
cmp [ebp+8], 3
ja @ret
mov eax, [ebp+8]
shl eax, 2
add eax, JMPTBL
mov eax, [eax]
jmp eax
ret

Symbolic Execution
(input:esp, ebp, memory)
push ebp
mov ebp, esp
cmp [esp+8], 3
ja @ret
mov ebp1 := esp
cmp [ebp1+8], 3
ja @ret
mov eax, [ebp+8]
shl eax, 2
add eax, JMPTBL
mov eax, [eax]
jmp eax
ret

Path predicate \( \varphi \):
@ebp1+8] < 3 \land eax4 == 2
@esp+8] < 3 \land \lbrack\lbrack(\lbrack\lbrackesp+8\rbrack\rbrack\ll 2) + JMPTBL] == 2
### DSE Vs Static & Dynamic approaches

#### Advantages:

- **sound program execution** (thanks to dynamic)
- **path sure to be feasible** (unlike static)
- **next instruction always known** (unlike static)
- **loops are unrolled by design** (unlike static)
- **can generate new inputs** (unlike dynamic)
- **guided new paths discovery** (unlike dynamic)
- **thwart basic tricks** (cover-overlapping etc)

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic</th>
<th>symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>✅</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>robust (obfuscation)</td>
<td>❌</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>correct</td>
<td>❌</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>complete</td>
<td>✅</td>
<td>❌</td>
<td>✅</td>
</tr>
</tbody>
</table>

The challenge for DSE is to make it scale on huge path length and to cover all paths...
3 Backward-Bounded DSE
Complementary approach for infeasibility-based problems
**BB-DSE:** Example of a call stack tampering

**Goal**
Checking that the return address cannot be tampered by the function

- **false negative**: miss the tampering *(too small bound)*
- **correct**: find the tampering
- **complete**: validate the tampering for all paths
Backward-Bounded DSE (new)

Infeasibility query: Query aiming at proving the infeasibility of some events or configuration. (while traditional SE performs feasibility requests (paths, values) to generate satisfying inputs)

Properties:
- backward approach
- solve infeasibility queries
- goal-oriented computation
- bounded reasoning
- bound modulable for the need

<table>
<thead>
<tr>
<th></th>
<th>(forward) DSE</th>
<th>bb-DSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>feasibility queries</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>infeasibility queries</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>scale</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Not FP/FN free, but very low rates
Combination

Intertwining Dynamic, Static and Symbolic
Combination: Principles

Goal: Enlarging a safe dynamic CFG by static disassembly guided by DSE to ensure a safer and more precise disassembly handling some obfuscation constructs.

The ultimate goal is to provide a semantic-aware disassembly based on information computed by symbolic execution.
Combination: Principles

Features:
- Enlarge partial CFG on genuine conditional jump
- Use dynamic jumps found in the dynamic trace
- Do not disassemble dead branch of opaque predicate
- Disassemble the target of tampered ret
- Do not disassemble the return site of tampered ret

Promising results 10 to 32% less instructions in obfuscated programs (with opaque predicates, call stack tampering).
5 BINSEC
Binsec platform architecture

**BINSEC**
- main binary analysis platform
- DSE, BB-DSE
  - static

**PINSEC**
- dynamic analysis instrumentation

**IDASEC**
- IDA plugin for result exploitation

Open source and available at:
- Binsec+Pinsec: http://binsec.gforge.inria.fr
- IDASec: https://github.com/RobinDavid/idasec
Pintool based on Pin 2.14-71313

Features:

- Generate a protobuf execution trace (with all runtime values)
- Can limitate the instrumentation time / space
- Working on Linux / Windows
- Configurable via JSON files
- Allow on-the-fly value patching
- Retrieve some function parameters on known library functions
- Remote control (prototype)
- Self-modification layer tracking

Still lacks many anti-debug countermeasures..
**Binsec** (main platform)

Features:
- Front-end: x86 (+simplification)
- Disassembly: linear, recursive, linear+recursive
- Static analysis: abstract interpretation

**Binsec/SE** (symbolic execution engine)

Features:
- generic C/S policy engine
- path selection for coverage (thanks Josselin 🙌)
- configurable via JSON file
- (basic) stub engine for library calls (+cdecl, stdcall)
- analysis implementation
- path predicate optimizations
- SMT solvers supported: Z3, boolector, Yices, CVC4

Many other DSE engines: Mayhem (ForAllSecure), Triton (QuarksLab), S2E, and all DARPA CGC challengers ....
Features:
- DBA decoding of an instruction
- reading an execution trace
- colorizing path taken
- dynamic disassembly (following the execution trace)
- triggering analyses via remote connection to Binsec
- exploiting the results depending of the analysis triggered

Goal:
- triggering analyses remotely from IDA and retrieving the results for post-processing
- leveraging Binsec features into IDA

Python plugin for IDA (from 6.4)
Packers study

Packers & X-Tunnel
Packer: deobfuscation evaluation

Evaluation of 33 packers (packed with a stub binary)

Looking for (with BB-DSE):
- Opaque predicates
- Call stack tampering
- record of self-modification layers

Settings:
- execution trace limited to 10M instructions

Goal: To perform a systematic and fully automated evaluation of packers
## Packer: Analysis results

<table>
<thead>
<tr>
<th>Packer</th>
<th>Trace len.</th>
<th>#proc</th>
<th>#th</th>
<th>#SMC</th>
<th>opaque predicates (OK)</th>
<th>opaque predicates (OP)</th>
<th>Call/stack tampering (OK)</th>
<th>Call/stack tampering (tamper)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACProtect v2.0</td>
<td>1.8M</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>83</td>
<td>159</td>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>ASPack v2.12</td>
<td>377K</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>168</td>
<td>24</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>Crypter v1.12</td>
<td>1.1M</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>399</td>
<td>24</td>
<td>125</td>
<td>78</td>
</tr>
<tr>
<td>Expressor</td>
<td>635K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>81</td>
<td>8</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>FSG v2.0</td>
<td>68k</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>24</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Mew</td>
<td>59K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>28</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>PE Lock</td>
<td>2.3M</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>95</td>
<td>90</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>RLPack</td>
<td>941K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>46</td>
<td>2</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>TELock v0.51</td>
<td>406K</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Upack v0.39</td>
<td>711K</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>41</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

- Several don’t have such obfuscation, NeoLite, nPack, Packman, PE Compact ....
- Several packers still evade the DBI, Armadillo, BoxedApp, EP Protector, VMProtect....
- 3 reached the 10M instructions limit, Enigma, svk, Themida
### Packer: Analysis results

<table>
<thead>
<tr>
<th>Packer</th>
<th>Trace len.</th>
<th>#proc</th>
<th>#th</th>
<th>#SMC</th>
<th>opaque predicates</th>
<th>Call/stack tampering</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACProtect v2.0</td>
<td>1.8M</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>159</td>
<td>0 48</td>
</tr>
<tr>
<td>ASPack v2.12</td>
<td>377K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>24</td>
<td>11 6</td>
</tr>
<tr>
<td>Crypter v1.12</td>
<td>1.1M</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>399</td>
<td>24 125 78</td>
</tr>
<tr>
<td>Expressor</td>
<td>635K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>81</td>
<td>8 14 0</td>
</tr>
<tr>
<td>FSG v2.0</td>
<td>68k</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>24</td>
<td>1 6 0</td>
</tr>
<tr>
<td>Mew</td>
<td>59K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>28</td>
<td>1 6 1</td>
</tr>
<tr>
<td>PE Lock</td>
<td>2.3M</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>95</td>
<td>90 4 3</td>
</tr>
<tr>
<td>RLPack</td>
<td>941K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>46</td>
<td>2 14 0</td>
</tr>
<tr>
<td>TELock v0.51</td>
<td>406K</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>2 3 1</td>
</tr>
<tr>
<td>Upack v0.39</td>
<td>711K</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>41</td>
<td>1 7 1</td>
</tr>
</tbody>
</table>

The technique scales on significant traces.

- Several don’t have such obfuscation, NeoLite, nPack, Packman, PE Compact ....
- Several packers still evade the DBI, Armadillo, BoxedApp, EP Protector, VMProtect....
- 3 reached the 10M instructions limit, Enigma, svk, Themida
### Packer: Analysis results

<table>
<thead>
<tr>
<th>Packer</th>
<th>Trace len.</th>
<th>#proc</th>
<th>#th</th>
<th>#SMC</th>
<th>opaque predicates</th>
<th>Call/stack tampering</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACProtect v2.0</td>
<td>1.8M</td>
<td>1</td>
<td></td>
<td></td>
<td>OK</td>
<td>159</td>
</tr>
<tr>
<td>ASPack v2.12</td>
<td>377K</td>
<td>1</td>
<td></td>
<td></td>
<td>OK</td>
<td>24</td>
</tr>
<tr>
<td>Crypter v1.12</td>
<td>1.1M</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>OK/Tamper</td>
<td>399</td>
</tr>
<tr>
<td>Expressor</td>
<td>635K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(OK)</td>
<td>24</td>
</tr>
<tr>
<td>FSG v2.0</td>
<td>68k</td>
<td>1</td>
<td></td>
<td></td>
<td>OK</td>
<td>125</td>
</tr>
<tr>
<td>Mew</td>
<td>59K</td>
<td>1</td>
<td></td>
<td></td>
<td>OK</td>
<td>6</td>
</tr>
<tr>
<td>PE Lock</td>
<td>2.3M</td>
<td>1</td>
<td></td>
<td>6</td>
<td>(OK)</td>
<td>90</td>
</tr>
<tr>
<td>RLPack</td>
<td>941K</td>
<td>1</td>
<td></td>
<td>1</td>
<td>OK</td>
<td>46</td>
</tr>
<tr>
<td>TELock v0.51</td>
<td>406K</td>
<td>1</td>
<td></td>
<td>5</td>
<td>(OK)</td>
<td>5</td>
</tr>
<tr>
<td>Upack v0.39</td>
<td>711K</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>(OK)</td>
<td>41</td>
</tr>
</tbody>
</table>

- Several don’t have such obfuscation, NeoLite, nPack, Packman, PE Compact ....
- Several packers still evade the DBI, Armadillo, BoxedApp, EP Protector, VMProtect....
- 3 reached the 10M instructions limit, Enigma, svk, Themida
Several don’t have such obfuscation, NeoLite, nPack, Packman, PE Compact ….

Several packers still evade the DBI, Armadillo, BoxedApp, EP Protector, VMProtect….  

3 reached the 10M instructions limit, Enigma, svk, Themida

### Packer: Analysis results

<table>
<thead>
<tr>
<th>Packer</th>
<th>Trace len.</th>
<th>#proc</th>
<th>#th</th>
<th>#SMC</th>
<th>opaque predicates (OK)</th>
<th>Call/stack tampering (OK)</th>
<th>Call/stack tampering (tamper)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACProtect v2.0</td>
<td>1.8M</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>159</td>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>ASPack v2.12</td>
<td>377K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>24</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>Crypter v1.12</td>
<td>1.1M</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>399</td>
<td>24</td>
<td>125</td>
</tr>
<tr>
<td>Expressor</td>
<td>635K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSG v2.0</td>
<td>68K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mew</td>
<td>59K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>28</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>PE Lock</td>
<td>2.3M</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>95</td>
<td>90</td>
<td>4</td>
</tr>
<tr>
<td>RLPack</td>
<td>941K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>46</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>TELock v0.51</td>
<td>406K</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upack v0.39</td>
<td>711K</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

The technique scales on significant traces

Many true positives. Some packers are using it intensively

Packers using ret to perform the final tail transition to the original entrypoint
**Packer: Tricks and patterns found**

**OP in ACProtect**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1018f7a</td>
<td>js</td>
<td>0x1018f92</td>
<td>(and all possible variants ja/jbe, jp/jnp, jo/jno..)</td>
</tr>
<tr>
<td>1018f7c</td>
<td>jns</td>
<td>0x1018f92</td>
<td></td>
</tr>
</tbody>
</table>

**OP in Armadillo**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10330ae</td>
<td>xor ecx, ecx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10330b0</td>
<td>jnz 0x10330ca</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CST in ACProtect**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001000</td>
<td>push 16793600</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001005</td>
<td>push 16781323</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100100a</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100100b</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CST in ASPack**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10043a9</td>
<td>mov [ebp+0x3a8], eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10043af</td>
<td>popa 0x10043bb</td>
<td>at runtime</td>
<td></td>
</tr>
<tr>
<td>10043b0</td>
<td>jnz 0x10043ba</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10043ba</td>
<td>push 0x10011d7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10043bf</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**OP (decoy) in ASPack**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10040fe</td>
<td>mov bl, 0x0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10041c0</td>
<td>cmp bl, 0x0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004103</td>
<td>jnz 0x1004163</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CST in ACProtect**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1004328</td>
<td>call 0x1004318</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004318</td>
<td>add [esp], 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100431c</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ZF = 0**

1004163: jmp 0x100416d

**ZF = 1**

1004105: inc [ebp+0xec]
X-Tunnel

A dive into the APT28 ciphering proxy
Introduction: Sednit / APT28 / Pawn Storm

Nicknames: APT28, Fancy Bear, Sofacy, Sednit, Pawn Storm

Alleged attacks:
- NATO, EU institutions  [2015]
- German Parliament (Germany)  [2015]
- TV5 Monde (France)  [2015]
- DNC: Democratic National Committee (US)  [2016]
- Political activists (Russia)
- MH17 investigation team (Netherlands)  [2015]
- Many more embassies and military entities ....

0-days used:
- 2 Flash  [CVE-2015-7645]
- 1 Office (RCE)  [CVE-2015-2424]
- 2 Java  [CVE-2015-2590]
- 1 Windows (LPE) [CVE-2015-4902]
  (delivered via their exploit kit “sedkit” with many existing exploits)

Tools used:
- Droppers / Downloader
- X-Agent / X-tunnel
- Rootkit / Bootkit
- Mac OS X trojan (Komplex)
- USB C&C

Data collected from: ESET, Trend Micro, CrowdStrike ...
X-Tunnel

What it is?
Ciphering proxy allowing X-Agent(s) not able to reach the C&C directly to connect to it through X-Tunnel.

Features
Encapsulate any TCP-based traffic into a RC4 cipher stream embedded into a TLS connection.

Samples

<table>
<thead>
<tr>
<th></th>
<th>Sample #0</th>
<th>Sample #1</th>
<th>Sample #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>42DEE3[...]</td>
<td>C637E0[...]</td>
<td>99B454[...]</td>
</tr>
<tr>
<td>Size</td>
<td>1.1 Mo</td>
<td>2.1 Mo</td>
<td>1.8 Mo</td>
</tr>
<tr>
<td>Creation date</td>
<td>25/06/2015</td>
<td>02/07/2015</td>
<td>02/11/2015</td>
</tr>
<tr>
<td>#functions</td>
<td>3039</td>
<td>3775</td>
<td>3488</td>
</tr>
<tr>
<td>#instructions</td>
<td>231907</td>
<td>505008</td>
<td>434143</td>
</tr>
</tbody>
</table>

A huge thanks to ESET Montreal and especially to Joan Calvet 😊

widely obfuscated with opaque predicates
Can we remove the obfuscation?

Are there new functionalities?
Can we remove the obfuscation?  

Are there new functionalities?

spoiler:
X-Tunnel: Analysis

Goal: Detecting and removing all opaque predicates to extract a clean CFG of the functions

Analysis context:
- full static analysis (because need to connect C2C, wait clients...)
- perform the backward-bounded DSE combined with IDA
- driven by IDASec

Combination divergence:
- without the dynamic component (ok because no SMC)
- the symbolic disassembly reduction performed “a-posteriori”

Analysis procedure:
1. opaque predicate detection
2. high-level predicate recovery
3. dead and spurious instruction removal
4. reduced CFG extraction

IDASec features used:
1. custom CFG structure to enumerate paths and which support annotation
2. liveness propagation
3. custom SMT formula
4. CFG extraction based on annotations
High-level predicate recovery (synthesis)

**Behavior:** Computes the dependency for a conditional jump, and recursively replace terms in order to obtain the predicate.

**Corollary:** The algorithm is able to determine which instructions are used for the computation of a conditional jump.

CFG

```
mov esi, dword_5D7A84
mov edi, dword_5D7A80
jz loc_44D9FA
imul esi, esi
imul eax, esi, 7
dec eax
imul edi, edi
cmp eax, edi
jnz loc_44D922
```

SMT Formula

```
(define-fun esi2 (load32_at memory #x005d7a84))
(define-fun edi0 (load32_at memory #x005d7a80))
(assert (not (= ZF2 #b1)))
(define-fun esi3 (bvmul (esi2 esi2)))
(define-fun eax2 (bvmul (esi3) #x00000007))
(define-fun eax3 (bvsub (eax2) #x00000001))
(define-fun edi1 (bvmul (edi0 edi0)))
(define-fun res328 (bvsub (eax3 edi1)))
(define-fun ZF4 (bvcomp res328 #x00000000))
(assert (= ZF4 #b1))

((bvsub (bvmul (bvmul esi2 esi2) #x7) #x1) ≠ (bvmul edi0 edi0)) → 7x^2 - 1 ≠ y^2
Analysis: Results

<table>
<thead>
<tr>
<th></th>
<th>#cond jmp</th>
<th>bb-DSE</th>
<th>Synthesis</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>C637 #1</td>
<td>34505</td>
<td>57m36</td>
<td>48m33</td>
<td>1h46m</td>
</tr>
<tr>
<td>99B4 #2</td>
<td>30147</td>
<td>50m59</td>
<td>40m54</td>
<td>1h31m</td>
</tr>
</tbody>
</table>

(only one path per conditional jump is analysed)

Only 2 different opaque predicate

\[ 7x^2 - 1 \neq x^2 \]

\[ \frac{2}{x^2 + 1} \neq y^2 + 3 \]

both present in the same proportions..
Analysis: Obfuscation distribution

Goal: Computing the percentage of conditional jump obfuscated within a function

Very few function are obfuscated ~500 (due to statically linked library not obfuscated OpenSSL etc..)

This allow nonetheless to **narrow the post-analysis on these functions** (likely of interest) ...
Analysis: Code coverage

Results of the liveness propagation and identification of spurious instructions

<table>
<thead>
<tr>
<th></th>
<th>C637 Sample #1</th>
<th>99B4 Sample #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Total instruction</td>
<td>505,008</td>
<td>434,143</td>
</tr>
<tr>
<td>#Alive</td>
<td>+279,483</td>
<td>+241,177</td>
</tr>
<tr>
<td>#Dead              -121,794</td>
<td>-113,764</td>
<td></td>
</tr>
<tr>
<td>#Spurious          -103,731</td>
<td>-79,202</td>
<td></td>
</tr>
<tr>
<td>#Delta with sample #0</td>
<td>47,576</td>
<td>9,270</td>
</tr>
</tbody>
</table>

In both samples the difference with the un-obfuscated binary is very low, and probably due to some noise.
Analysis: Reduced CFG extraction

Goal: Performing a-posteriori the static disassembly sketch in the combined approach

Algorithm:
- remove basic blocks marked dead
- remove spurious instructions (part of the computation of OP)
- recreate the CFG by concatenating instructions with a single predecessor

Result:

Original CFG → CFG marked → CFG extracted
Demo!

X-Tunnel deobfuscation
X-Tunnel: Conclusion

Manual checking of difference to not appeared to yield significant differences or any new functionalities...

Obfuscation: Differences with O-LLVM (like)
- some predicates have a great dependency (use local variables)
- some computation reuse between opaque predicates

Technique:
- Combination: Backward Symbolic Execution and “a-posteriori” static disassembly reduction (without the dynamic aspect)
- very few FP / FN refined manually by predicate synthesized (due to the low diversity of predicates)

Next:
- in-depth graph similarity (to find new functionalities)
- integration as an IDA processor module (IDP) ?

For more: Visiting the Bear Den
Joan Calvet, Jessy Campos, Thomas Dupuy

[RECON 2016][Botconf 2016]
Binsec Takeaways

Tip of what can be done with Binsec:
- dynamic symbolic execution,
- abstract interpretation,
- simulation,
- optimizations,
- simplifications,
- on-the-fly value patching...

More is yet to come:
- documentation,
- ARMv7 support,
- code flattening and VM deobfuscation...

Still a young platform:
- under heavy development,
- API not stabilized,
  (considering rewriting IDASec with Binary Ninja)...

Take part!
- Download it, try it, experiment it!
- Don’t hesitate contacting us for questions!

Open source and available at:
- Binsec+Pinsec: http://binsec.gforge.inria.fr
- IDASec: https://github.com/RobinDavid/idasec
Takeaways

- **More is not always better** in terms of disassembly on obfuscated programs.

- The backward bounded DSE scale well and allowed to detect obfuscations considered on many packers and X-Tunnel.

- The combination yielded very good results on X-Tunnel.

- The combination dynamic, static and symbolic is the way to go on obfuscated binaries and helped recovering a clean CFG on X-Tunnel. Still under integration in Binsec with support of different self-modification layers....
Thank you!

Q & A

Robin David
robin.david@riseup.net
@RobinDavid1

Sébastien Bardin
sebastien.bardin@cea.fr