Reveal Secrets in Adoring Poitras
A generic attack on white-box cryptography

Junwei Wang
CryptoExperts
University of Luxembourg
University of Paris 8
ECRYPT-NET School on Correct and Secure Implementation
October 11, 2017, Crete
Outline

1. White-Box Cryptography
   - What Is White-Box Cryptography (WBC)?
   - WhiBox Contest

2. Breaking Adoring Poitras
   - Cleaning the Code
   - De-Virtualization
   - Bitwise-Based Program to Boolean Circuits
   - Boolean Circuits Minimization
   - Data Dependency Analysis
   - Algebraic Analysis
What Is White-Box Cryptography (WBC)?

- WBC is resistant against **key extraction** in a **software** implementation of a cryptographic algorithm.
- The attacker **entirely** controls the running environment.
  - to record the computation trace (memory address/value, access type/time, etc)
  - to modify the control flow / intermediate value, etc
- No provably secure construction exists.
- All known practical constructions has been broken by **generic attacks** (DCA and DFA) before 2016.
- Applications:
  - digital rights management (DRM)
  - mobile payments
WhiBox Contest - CHES 2017 CTF

- Organized by ECRYPT CSA
- Two categories:
  - designers
  - breakers
- AES-128, physical limitation (<50M source code, <20M binary, <1s execution)
- 94 submitted challenges are all broken (most of them were alive < 1 day)
  - Surviving for 28 days (2.3 × the 2nd hardest one)
  - Submitted by cryptolux (Biryukov-Udovenko)
  - Only broken by team_cryptoeXperts (Goubin-Paillier-Rivain-Wang)
Untidy Code

More than 1k functions

- Random naming
- Not used
- Duplicate
Readability Processing

- Duplicate / redundancy / unused codes elimination
- Functions / variables renaming
- Constants rewriting
- Code combination

Only 20 functions are remaining
Universal Turing Machine

\[ \Rightarrow \text{UTM}(\text{RASP}) \]
Universal Turing Machine (2)

void called() {
    unYAQ = klspCVy;
    sutlnu = klspCVy + sizeof(JGNNvi)/sizeof(uchar);
    JqcadL = klspCVy;
    while (JqcadL < sutlnu) {
        uchar eMnr = *JqcadL++;
        if (eMnr == 0) {
            void (*QIEb)();
            QIEb = (void*)funcptrs[*JqcadL++];
            uint *AnezSv = (uint*)JqcadL;
            JqcadL += eMnr*8;
            QIEb();
        } else if (eMnr == 1) {
            void (*QIEb)(uint);
            QIEb = (void*)funcptrs[*JqcadL++];
            uint *AnezSv = (uint*)JqcadL;
            JqcadL += eMnr*8;
            QIEb(AnezSv[0]);
        }
    } ...
}

void AES_128_encrypt(uchar *OLjd, uchar *xzptIF) {
    for(kIKfgI = 0; kIKfgI < sizeof(ooGoRv)/sizeof(uint); kIKfgI++) ooGoRv[kIKfgI] = gBXW[kIKfgI];
    xkpRp = 0Ljd;
    Puix = xzptIF;
    if (sizeof(klspCVy))
        #LL();
    else
        #.; // c;
}
Universal Turing Machine (3)
De-virtualization - Simulate the UTM

We get a bitwise-based program (600k operations).
Bitwise-Based Program

Input: plaintext bits \((b_1, b_2, \cdots, b_{128})\)
Output: ciphertext bits \((c_1, c_2, \cdots, c_{128})\)

\[
\begin{align*}
\text{for } i &= 1 \text{ to } 128 \text{ do} \\
&\quad t[addr_1,i] \leftarrow 0bb_i\cdots b_i \\
&\quad \text{for } j = 1 \text{ to } 64 \text{ do} \\
&\quad \quad t[addr_2,i + j \cdot 2^{12}] \leftarrow t[addr_1,i] \\
&\quad \text{end for} \\
&\quad \text{end for} \\
&\text{BitwiseOperationLoop1} \\
&\text{BitwiseOperationLoop2} \\
&\quad \quad \cdots \\
&\text{BitwiseOperationLoop2573} \\
\text{for } i &= 1 \text{ to } 129 \text{ do} \\
&\quad t[addr_3,i] \leftarrow v_i \\
&\quad \text{for } j = 1 \text{ to } 64 \text{ do} \\
&\quad \quad tmp \leftarrow t[addr_4,i + j \cdot 2^{12}] \oplus t[addr_5,i + j \cdot 2^{12}] \\
&\quad \quad t[addr_3,i] \leftarrow t[addr_3,i] \oplus \text{Parity}(tmp) \\
&\quad \text{end for} \\
&\quad \text{end for} \\
&\text{BitwiseOperationLoop2574} \\
&\quad \quad \cdots \\
&\text{BitwiseOperationLoop2582} \\
\text{for } i &= 1 \text{ to } 128 \text{ do} \\
&\quad c_i \leftarrow t[addr_6,i] \\
&\text{end for}
\end{align*}
\]

▷ expand \(b_i\) to unsigned long integer (64 bits)
▷ loop for 64 times
▷ \(v_i \in \text{GF}(2)\) is a constant
▷ \text{Parity} computes the number of 1-bit modulo 2
Bitwise-Based Program to Boolean Circuits

- 64 (loop length) * 64 (number of bits in a unsigned long integer) independent AES computations operated in boolean circuits
- 3 out of 64*64 are the real and identical AES computations (e.g., bit 42 of loop 26)
- Hence, the bitwise-based program can be simplified as a boolean circuits with 600k gates (XOR, AND, OR, NOT).

Breakers are stopped by this step??
Boolean Circuits Minimization

- Constant variable detection and propagation
- Deduplication
- “Potential” pseudorandomness detection and removal
- Dead code elimination
- Repeat the above steps until no more constant / duplicate / "potential" pseudorandomness can be detected

The circuits is reduced to 280k boolean gates (53% smaller)
Data Dependency Graph (DDG)

\[ x = a; \]
\[ y = b; \]
\[ x = y + x; \]
\[ y = x \times y; \]
\[ z = x - y; \]
\[ x = z \times x; \]
DDG of the Circuits (First 5%)
First Round Computation of AES

MixColumns
SubBytes
Extracting the Branches (Clustering)
Assumption

Assumption (Informal)

Each of the green ”branch” corresponds to an individual S-Box computation in the first round of AES, the $t$-bit output $(s_1, s_2, \ldots, s_t)$ of which is a linear encoding of a real S-Box output bit.
Output Bits of A Branch

Bits in a branch (530)

S-Box output bits (34)
Solve A System of Linear Equations

\[
\begin{bmatrix}
  s_1^{(1)} & s_2^{(1)} & \ldots & s_{34}^{(1)} & 1 \\
  s_1^{(2)} & s_2^{(2)} & \ldots & s_{34}^{(2)} & 1 \\
  \vdots & \vdots & \ddots & \vdots & \vdots \\
  s_1^{(n)} & s_2^{(n)} & \ldots & s_{34}^{(n)} & 1
\end{bmatrix}
\begin{bmatrix}
  a_1 \\
  a_2 \\
  \vdots \\
  a_{34} \\
  a_{35}
\end{bmatrix}
= 
\begin{bmatrix}
  \text{SBox}(x^{(1)} \oplus \hat{k})[i] \\
  \text{SBox}(x^{(2)} \oplus \hat{k})[i] \\
  \vdots \\
  \text{SBox}(x^{(n)} \oplus \hat{k})[i]
\end{bmatrix}
\]

If \( n \geq 35 + 8 + \lambda \), \( \Pr[\text{"}\hat{k} \neq k^* \text{ has a solution"}] \leq 2^{-\lambda} \).
Results

```
in[48] = LinearBreak[data]

key=0x0
key=0x10
key=0x20
key=0x30
key=0x40
key=0x50
key=0x60
key=0x70
key=0x80
key=0x90
key=0xA0
key=0xB0
key=0xc0

!!!!!!!!!!!!!!!! 2  -  0  -  8xcf  !!!!!!!!!!!!!
!!!!!!!!!!!!!!!! 2  -  1  -  8xcf  !!!!!!!!!!!!!
!!!!!!!!!!!!!!!! 2  -  2  -  8xcf  !!!!!!!!!!!!!
!!!!!!!!!!!!!!!! 2  -  3  -  8xcf  !!!!!!!!!!!!!
!!!!!!!!!!!!!!!! 2  -  4  -  8xcf  !!!!!!!!!!!!!
!!!!!!!!!!!!!!!! 2  -  5  -  8xcf  !!!!!!!!!!!!!
!!!!!!!!!!!!!!!! 2  -  6  -  8xcf  !!!!!!!!!!!!!
!!!!!!!!!!!!!!!! 2  -  7  -  8xcf  !!!!!!!!!!!!!
key=0xd0
key=0xe0
key=0xf0
```
Why DCA / DFA does not work?

0: \{0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}
1: \{0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}
2: \{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}
3: \{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}
4: \{0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}
5: \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}
6: \{0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}
7: \{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}

15 used / 34 output bits
Why DCA / DFA does not work?

0: \[\{0,0,0,0,0,1,0,1,0,1,1,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\}\]
1: \[\{0,0,0,0,0,1,0,0,1,1,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\}\]
2: \[\{0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,1,1,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0\}\]
3: \[\{0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0\}\]
4: \[\{0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,0,0,0,0,0,1,0,1,1,1,0,0,0,0,0,0,0,0,0\}\]
5: \[\{0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\}\]
6: \[\{0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0\}\]
7: \[\{0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\}\]

Each real bit is encoded by at least 2 intermediate bits.
Why DCA / DFA does not work?

Each intermediate bit is encoding at least for 2 real output bits.
Summary and Future Works

- White-box cryptography is widely deployed.
- All known constructions are broken by DFA and DCA attacks before 2016.
- A algebraic analysis attack is applied to break challenges.

Future works:
- Countermeasures to design
- Generalization of this attack
- Theoretical construction
Thank you!

Question?