Another Look on Bucketing Attack to Defeat White-Box Implementations

Mohamed Zeyad[2*], Houssem Maghrebi[1],
Davide Alessio[1], and Boris Batteux[3*]
[1] UL Identity Management & Security, France
firstname.lastname@ul.com
[2] Trusted Labs, France
mohamed.zeyad@trusted-labs.com
[3] Eshard, France
boris.batteux@eshard.com

4th April 2019, COSADE 2019, Darmstadt

[*] This work was done while the authors were working at UL Identity Management & Security

AGENDA

1. Context and Motivation
2. Statistical Bucketing attack
3. Computational Bucketing attack
4. Comparison with other similar attacks
5. Experimental results
6. Conclusions

AGENDA

CONTEXT

White-box cryptography was introduced in 2002 by Chow, Eisen, Johnson and van Oorschot

White-box cryptography is about implementations of cryptographic algorithms to be executed in an environment completely under the adversary control, i.e.: the white-box model

The white-box model is the scenario closest to the real-world capturing the idea of an adversary with full knowledge and full control of the targeted device

MOTIVATION

Protect illegitimate access to copyrighted content from unauthorized users (DRM)

Applications managing sensible and personal data run on untrusted devices like smartphone and portable devices (smartwatches, …)

Firmware with high intellectual property value runs on non-secure hardware (IoT)

Protect access, banking and transit assets in software-only representations (HCE, CBP)

WHITE-BOX IMPLEMENTATION

Chow *et al.* introduced the first design of white-box implementations for the DES and AES ciphers:

1. Represent the algorithm as a network of look-up tables
2. Randomize tables and their inputs/outputs with random encodings
3. *Glue* the white-box implementation to the surrounding software with random external encodings

Several designs have been published with approaches similar to this framework

CHOW ET AL.'S AES ORIGINAL DESIGN

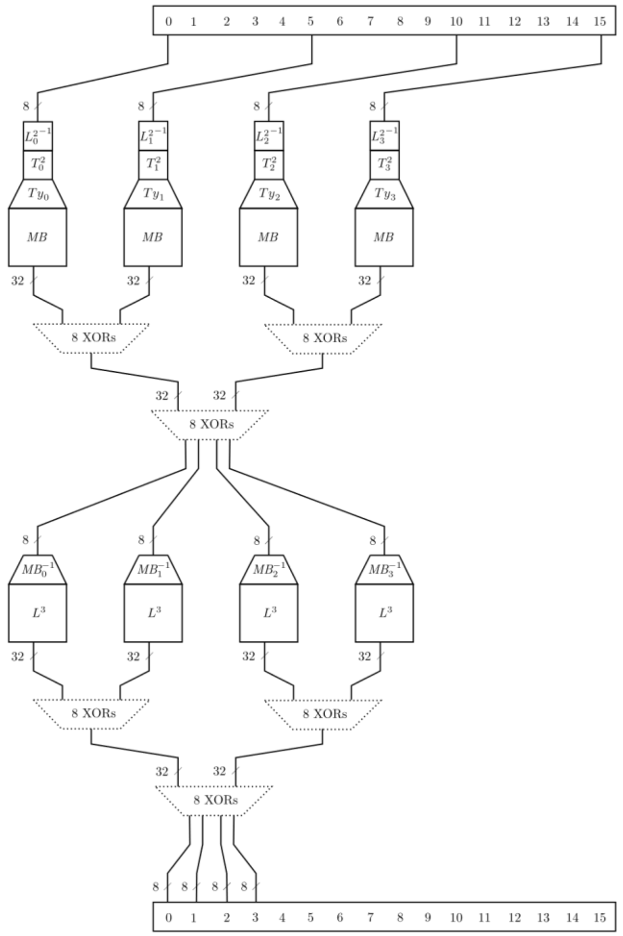
A cipher state row is computed as:

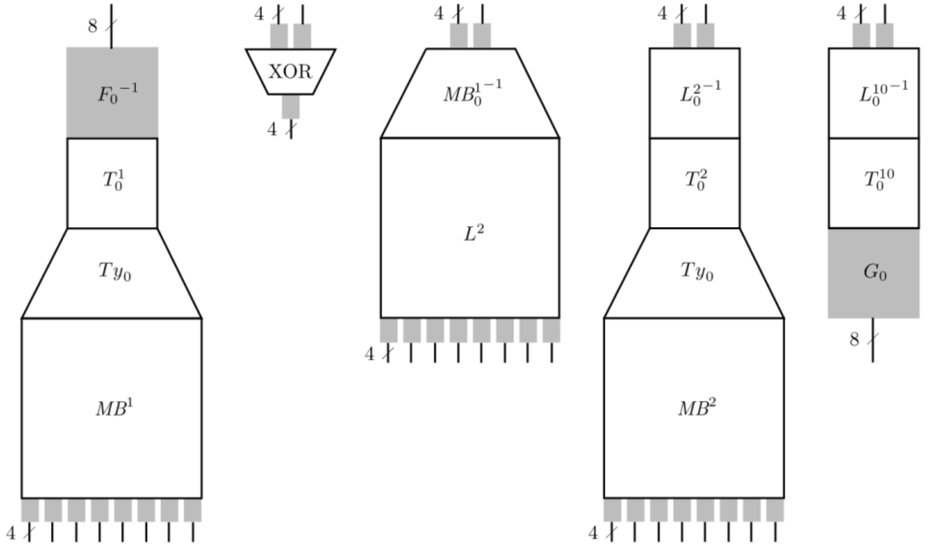
$$(y_0, y_1, y_2, y_3) = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \otimes \begin{pmatrix} S(x_0) \oplus k_0 \\ S(x_5) \oplus k_5 \\ S(x_{10}) \oplus k_{10} \\ S(x_{15}) \oplus k_{15} \end{pmatrix}$$

that can be rearranged as follow:

$$(y_0, y_1, y_2, y_3) = T_{y_o}[x_0] \oplus T_{y_5}[x_5] \oplus T_{y_{10}}[x_{10}] \oplus T_{y_{15}}[x_{15}]$$

with tables $T_i[\cdot]: \{0 \dots 2^8\} \rightarrow \{0 \dots 2^{32}\}$ and $\oplus: \{0 \dots 2^8\} \rightarrow \{0 \dots 2^4\}$





Pictures from Muir "A Tutorial on White-box AES" (ePrint 2013, ia.cr/2013/104)

ATTACKS ON WHITE-BOX IMPLEMENTATIONS

Attacks can be divided into two families:

- Differential and algebraic cryptanalysis:
  - Recover the encodings, invert them and recover the key
    - BGE attack (*Billet et al.* SAC04)
    - BGE attack improved by *Lepoint et al.* at SAC13
    - Collisions attack (*Lepoint et al.* at SAC13)
- *Physical* attacks:
  - Differential computational analysis, fault injection analysis
    - Fault attack (*Sanfelix et al.* BlackHat15)
    - Differential Computational attack (*Bos et al.* CHES16)

AGENDA

STATISTICAL BUCKETING ATTACK

Cryptanalysis introduced by Chow *et al.* on the naked version of WB DES:

- Chosen-plaintext attack
- Key-recovery

Requirements:

- Access to some intermediate values
- Access to an encryption (or decryption) oracle

Output:

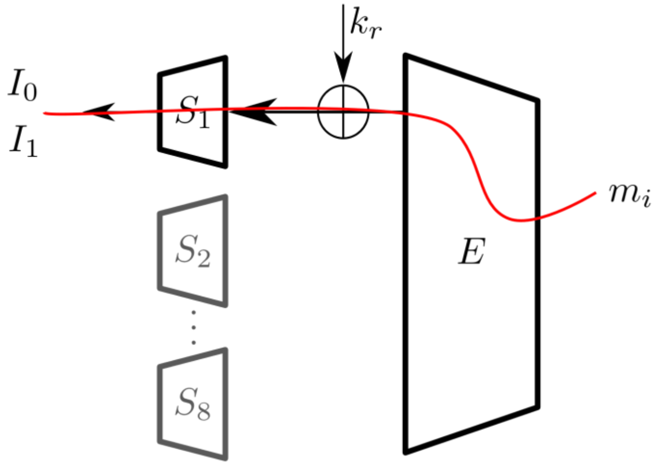
- 48 bits of the key (+ 8 bits to be brute-forced)

STATISTICAL BUCKETING ATTACK ALGORITHM

1. Select i^{th} Sbox on 1st round S_i^1
2. Guess 6 bits of the first round key and generate 2^6 possible plaintexts m
   i. Left-side of the message after permutation P are constant
   ii. Remaining 26 bits of right-side are chosen at random
3. Select one *bucketing* bit b of $S_i^1(\cdot)$ and divide plaintexts into two sets:
   $I_b = \{m | b \leftarrow S_i^1(E(m) \oplus k)\}$ with $b = 0, 1$
4. Select z^{th} Tbox on 2nd round T_z^2 encoding the *bucketing* bit and group its input in two sets $V_b = input_{T_z^2}\ (\mathrm{I_b})$ with $b = 0, 1$
5. Check if $V_0 \cap V_1 = \emptyset$ to confirm the 6-bit key guess, else reject
6. Repeat for S_j^1 with $j = 0, \ldots, 8$

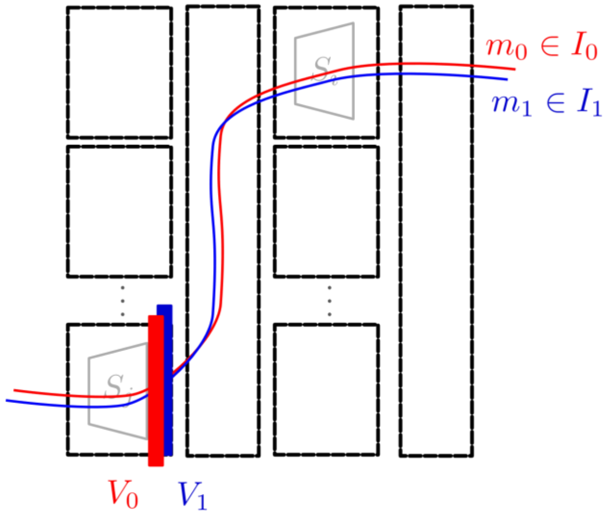
STATISTICAL BUCKETING ATTACK ALGORITHM

1. Select i^{th} Sbox on 1st round S_i^1
2. Guess 6 bits of the first round key and compute 2^6 possible plaintexts m
   i. Left-side of the message after permutation P are constant
   ii. Remaining 26 bits of right-side are chosen at random
3. Select one *bucketing* bit b of $S_i^1(\cdot)$ and divide plaintexts into two sets:
   $$I_b = \{m | b \leftarrow S_i^1(E(m) \oplus k)\} \text{ with } b = 0, 1$$
4. Select zth Tbox on 2nd round T_z^2 encoding the *bucketing* bit and group its input in two sets $V_b = input_{T_z^2}(\mathrm{I_b})$ with $b = 0, 1$
5. Check if $V_0 \cap V_1 = \emptyset$ to confirm the 6-bit key guess, else reject
6. Repeat for S_j^1 with $j = 0, \ldots, 8$



STATISTICAL BUCKETING ATTACK ALGORITHM

1. Select i^{th} Sbox on 1st round S_i^1
2. Guess 6 bits of the first round key and compute 2^6 possible plaintexts m
   i. Left-side of the message after permutation P are constant
   ii. Remaining 26 bits of right-side are chosen at random
3. Select one *bucketing* bit b of $S_i^1(\cdot)$ and divide plaintexts into two sets:
   $I_b = \{m|b \leftarrow S_i^1(E(m) \oplus k)\}$ with $b = 0, 1$
4. Select z^{th} Tbox on 2nd round T_z^2 encoding the *bucketing* bit and group its input in two sets $V_b = input_{T_z^2}(\mathrm{I_b})$ with $b = 0, 1$
5. Check if $V_0 \cap V_1 = \emptyset$ to confirm the 6-bit key guess, else reject
6. Repeat for S_j^1 with $j = 0, \ldots, 8$



STATISTICAL BUCKETING ATTACK ALGORITHM

1. Select i^{th} Sbox on 1st round S_i^1
2. Guess 6 bits of the first round key and compute 2^6 possible plaintexts m
   1. Left-side of the message after permutation P are constant
   2. Remaining 26 bits of right-side are chosen at random
3. Select one *bucketing* bit b of $S_i^1(\cdot)$ and divide plaintexts into two sets:
   $$I_b = \{m | b \leftarrow S_i^1(E(m) \oplus k)\} \text{ with } b = 0, 1$$
4. Select z^{th} Tbox on 2nd round T_z^2 encoding the *bucketing* bit and group its input in two sets $V_b = input_{T_z^2}(\mathrm{I_b})$ with $b = 0, 1$
5. Check if $V_0 \cap V_1 = \emptyset$ to confirm the 6-bit key guess, else reject
6. Repeat for S_j^1 with $j = 0, \ldots, 8$

$$V_0 \cap V_1 = \begin{cases} \emptyset & \rightarrow \text{🔑 ✓} \\ \{v_i\}_i & \rightarrow \text{🔑 ✗} \end{cases}$$

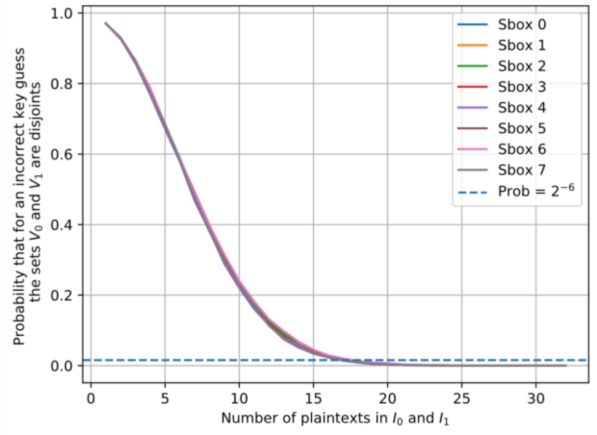
IS IT A GOOD DISTINGUISHER

If the key guess is correct, $V_0 \cap V_1 = \emptyset$ because their elements are different ***at least*** on bit b.

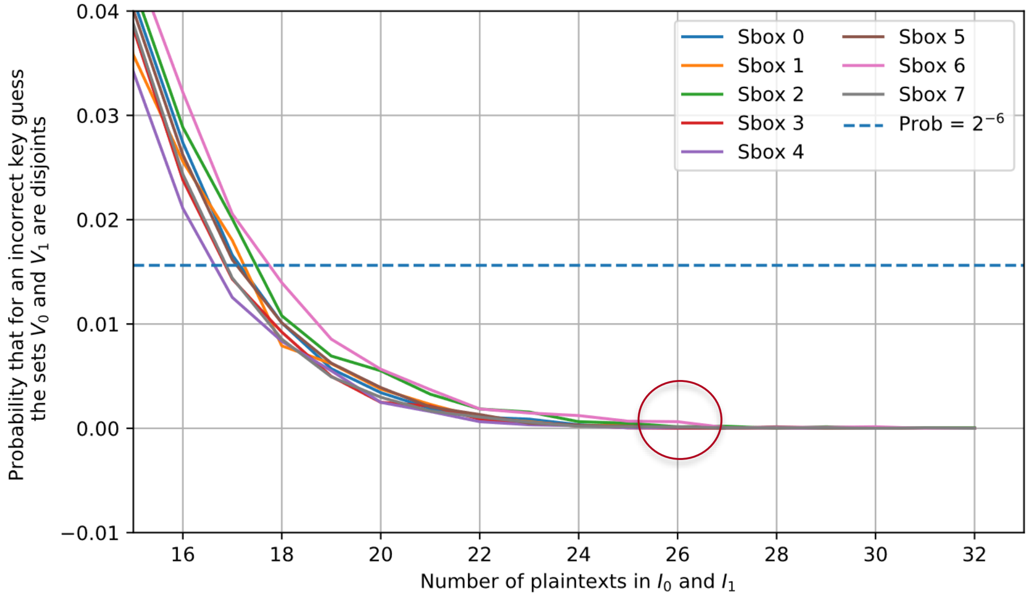
So: k is correct $\Longrightarrow V_0 \cap V_1 = \emptyset$

Are there false-positives?

EFFECTIVENESS OF THE SB DISTINGUISHER (DES)



Evolution of the probability that for an incorrect key guess the sets V_0 and V_1 are disjoint according to an increasing number of plaintexts



OUR CONTRIBUTION

- Extension of the Statistical Bucketing Attack to AES
- New automated key-recovery computational attack based on an algebraic cryptanalysis, the Statistical Bucketing Attack
- Validate our proposal through practical experiments
- Compare the efficiency of our attack w.r.t. the existing computational attacks

OUR EXTENSION TO AES

As is, Statistical Bucketing Distinguisher cannot work for AES:

$$I_0 = \{X_i | 0 \leftarrow S(X_i \oplus k)\}, \qquad I_1 = \{Y_j | 1 \leftarrow S(Y_j \oplus k)\}$$

Given a Tbox T_z^2 we define

$$V_0 = \{input_{T_z^2}(X_i)\}_{X_i \in I_0}, \qquad V_1 = \{input_{T_z^2}(Y_j)\}_{Y_j \in I_1}$$

And as both encodings and AES round function are bijections:

$$X_i \neq Y_j \Leftrightarrow E\left(L\left(S(X_i \oplus k)\right)\right) \neq E\left(L\left(S(Y_i \oplus k)\right)\right) \ \forall k$$

for $i, j = 0, \ldots, 2^7 - 1 \Rightarrow V_0 \cap V_1 = \emptyset$

OUR EXTENSION TO AES

As is, SBA cannot work for AES because encodings and AES round function are bijections:

$$X_i \neq Y_j \Leftrightarrow E\left(L\left(S(X_i \oplus k)\right)\right) \neq E\left(L\left(S(Y_i \oplus k)\right)\right)\ \forall k$$

$$\text{for } i, j = 0, \dots, 2^7 - 1 \Rightarrow V_0 \cap V_1 = \emptyset$$

IDEA

"… *replace the AES Sbox* with a non bijective one"

$$S' : GF(2^8) \rightarrow GF(2^4)$$
$$x \mapsto AESSbox(x)\ \&\ 0xf$$

Assumption: we focused on nibble-encoded white-box implementations

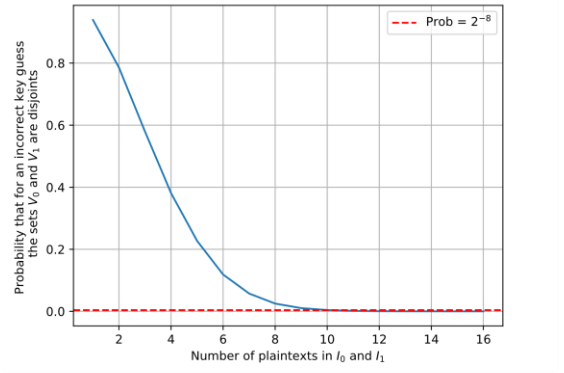
STATISTICAL BUCKETING ALGORITHM (AES)

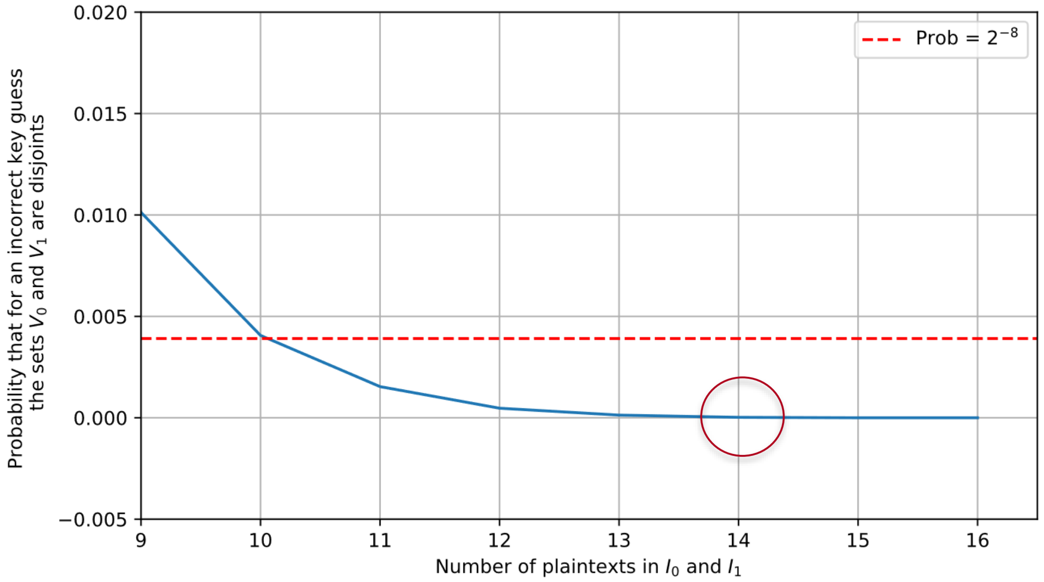
1. Select i^{th} Sbox on 1st round S_i^1
2. Guess 8 bits of the first round key and compute 2^8 possible plaintexts m
    - *i.* i^{th} byte in $\{0, \dots, 2^8\}$
    - ii. Remaining 15 bytes are constant
3. Select two distinct *bucketing nibbles* d_0 and d_1 and divide plaintexts into two sets:

$$I_b = \{m | d_b \leftarrow S'^1_i(m \oplus k)\} \text{ with } b = 0, 1$$

4. Select z^{th} Tbox on 2nd round T_z^2 encoding the *bucketing* nibble and group its input in two sets $V_b = input_{T_z^2}\,(\mathrm{I_b})$ with $b = 0, 1$
5. Check if $V_0 \cap V_1 = \emptyset$ to confirm the 8-bit key guess, else reject
6. Repeat for S_j^1 with $j = 0, \dots, 16$

EFFECTIVENESS OF THE SBA DISTINGUISHER (AES)





Evolution of the probability that for an incorrect key guess the sets V_0 and V_1 are disjoint according to an increasing number of plaintexts

AGENDA

BUCKETING COMPUTATIONAL ATTACK

Statistical bucketing attack is hard because of software security measures

$\Downarrow$

Exploit computational execution traces

- Instead of targeting Tbox *inputs,* we exploit the *execution traces*
- $\boldsymbol{V_0}$ and $\boldsymbol{V_1}$ are seen as (m, n)-matrices
- Bucketing distinguisher applied on each column:
    1. Count $\#\{j | \boldsymbol{V_0}[j] \cap \boldsymbol{V_1}[j] = \emptyset \textit{ for } j \in [0, n]\}$
    2. Output the key maximizing the cardinality

ALGORITHM (NIBBLE ENCODED)

Precomputation:

i. generate 2^8 messages targeting a Sbox

ii. chose two values $0 \leq d_0 < d_1 \leq 15$ (*bucketing nibbles*)

iii. for each key guess k:

group messages in sets $I_b = \{m | d_b \leftarrow S'^{1}_{i}(m \oplus k)\}$ with $b = 0, 1$

Output: 256 pairs of sets $\left(I_{0,k}, I_{1,k}\right)$ where each set contains 16 messages

ALGORITHM (NIBBLE ENCODED)

Precomputation:

i. generate 2^8 messages targeting a Sbox

ii. chose two values $0 \leq d_0 < d_1 \leq 15$

iii. for each key guess k:

group messages in sets $I_b = \{m | d_b \leftarrow S'^{1}_{i}(m \oplus k)\}$ with $b = 0, 1$

Output: 256 pairs of sets $\left(I_{0,k}, I_{1,k}\right)$ where each set contains 16 messages

Acquisition:

i. acquire a set of 256 traces $T = (t_{i,j})_{\substack{0 \leq i \leq 255 \\ 0 \leq j \leq n}}$ (encryption of $m \in I$ of n samples)

ALGORITHM (NIBBLE ENCODED)

Key recovery:

```
i.   initialize vector $R = [0, \ldots, 0]$ with 256 zeroes
ii.  for each key guess $k$:
         group traces in two matrices $\boldsymbol{V_{0,k}}$ and $\boldsymbol{V_{1,k}}$ according to sets $I_{0,k}, I_{1,k}$
         for each sample $0 \leq j \leq n$:
             if $\boldsymbol{V_{0,k}}[j] \cap \boldsymbol{V_{1,k}}[j] = \emptyset$:
                 $R[k] = R[k] + 1$
iii. output $k$ s.t. $R[k] = \max\{R[j]\}_j$
```

AGENDA

COMPARISON WITH OTHER ATTACKS

BCA can be seen as the chosen-plaintext counterpart of DPA: a key portion guess is confirmed or rejected by looking at some sets intersection, it is effective on software implementations because traces are noise-free

BCA is somewhat similar to ZDE, both uses sets of *well-chosen plaintexts* and perform a statistical analysis on the intermediate values

BCA is similar to the *collision-attack*, both look for collisions on internal encoded values, BCA imposes constraints on the input sets while *collision-attacks* verifies them later

AGENDA

EXPERIMENTAL RESULTS

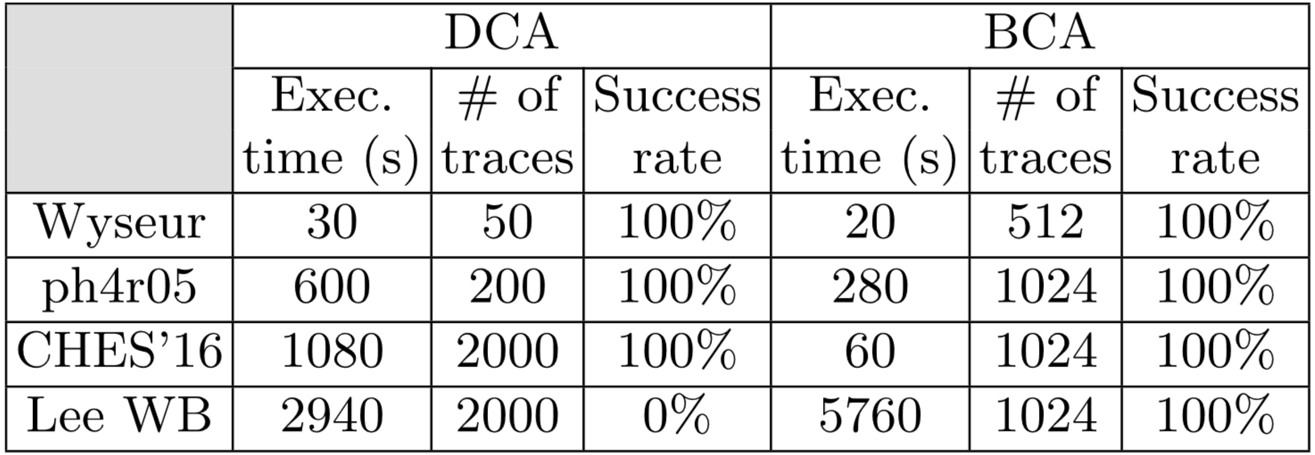
We run experiments on 4 public white-box implementations:

- DES: *Wyseur 2007* challenge
- AES: *ph4r05* implementing dual-ciphers variant by Karroumi
- AES: *CHES 2016* challenge
- AES: *Lee et al.*'s implementation (CASE 1)

All these challenges are based on *nibble-encoded* designs

Our tool is available on Github (https://github.com/Bucketing/BCA-attack)

EXPERIMENTAL RESULTS



	DCA			BCA		
	Exec. time (s)	# of traces	Success rate	Exec. time (s)	# of traces	Success rate
Wyseur	30	50	100%	20	512	100%
ph4r05	600	200	100%	280	1024	100%
CHES'16	1080	2000	100%	60	1024	100%
Lee WB	2940	2000	0%	5760	1024	100%

Note: We didn't run the ZDE attack because of the high number of required traces (500×2^{17} to recover two bytes of the key in the CHES'16 challenge)

COMPARISONS

- On CHES'16 running DCA is quite complicated because of the countermeasures and one needs to run DCA twice to recover the key; while BCA recovers the key faster
- On *Lee et al.'s* design DCA does not work, while BCA is efficient

On the other hand:

- On not protected implementations DCA is more efficient to fully recover the key (*Wyseur* and *ph4r05 challenges*)

COUNTERMEASURES

The Computational Bucketing Distinguisher is highly dependent on traces synchronization, therefore to protect the implementation one could:

- introduce misalignment on the execution traces (introducing dummy computations or shuffling the order of the operations)
- introduce dummy encryptions with dummy non-constant keys
- implement a proper masking

Some countermeasures would probably be defeated by reverse engineering on the targeted implementation

AGENDA

CONCLUSIONS

- Presented a new computational analysis method to break white-box implementations
- Proposed a new key distinguisher for automated computational attack
- Applied this cryptanalysis technique to AES

- Facing an unprotected implementation, DCA remains the most effective attack, our proposal becomes an good alternative facing an implementation with some countermeasures beating the complexity of LDA and HO-DCA

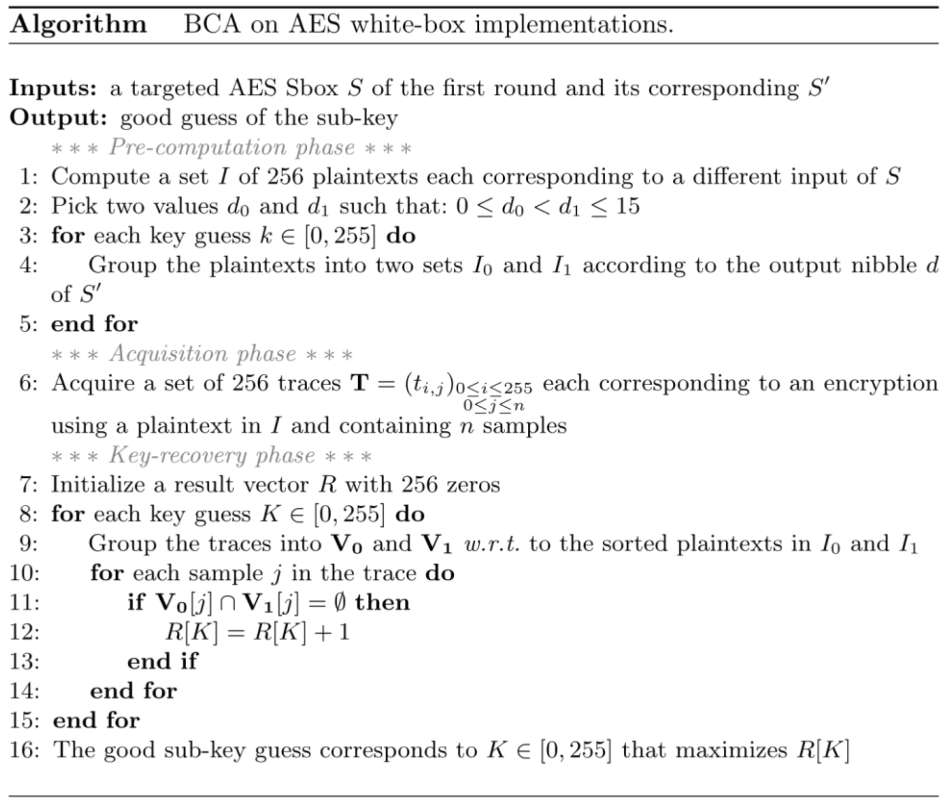
As future work we plan to expand the Computational Bucketing Distinguisher to an higher order context and to apply the attack to more complex implementations

Thank you

Questions??

Backup Slides

FULL ALGORITHM



Algorithm BCA on AES white-box implementations.

```
Inputs: a targeted AES Sbox $S$ of the first round and its corresponding $S'$
Output: good guess of the sub-key
    *** Pre-computation phase ***
 1: Compute a set $I$ of 256 plaintexts each corresponding to a different input of $S$
 2: Pick two values $d_0$ and $d_1$ such that: $0 \leq d_0 < d_1 \leq 15$
 3: for each key guess $k \in [0, 255]$ do
 4:     Group the plaintexts into two sets $I_0$ and $I_1$ according to the output nibble $d$ of $S'$
 5: end for
    *** Acquisition phase ***
 6: Acquire a set of 256 traces $\mathbf{T} = (t_{i,j})_{\substack{0 \leq i \leq 255 \\ 0 \leq j \leq n}}$ each corresponding to an encryption using a plaintext in $I$ and containing $n$ samples
    *** Key-recovery phase ***
 7: Initialize a result vector $R$ with 256 zeros
 8: for each key guess $K \in [0, 255]$ do
 9:     Group the traces into $\mathbf{V_0}$ and $\mathbf{V_1}$ w.r.t. to the sorted plaintexts in $I_0$ and $I_1$
10:     for each sample $j$ in the trace do
11:         if $\mathbf{V_0}[j] \cap \mathbf{V_1}[j] = \emptyset$ then
12:             $R[K] = R[K] + 1$
13:         end if
14:     end for
15: end for
16: The good sub-key guess corresponds to $K \in [0, 255]$ that maximizes $R[K]$
```