Generating Bitslice Implementations of Arbitrary S-Boxes

Augustin Bariant, Matthieu Daumas and Adrien Guinet

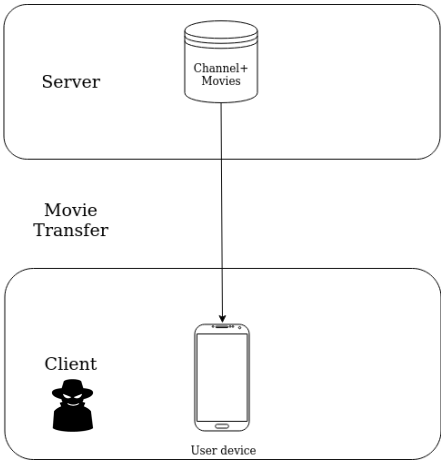
Quarkslab

Introduction: The Whitebox model

Whitebox: The attacker has **full access** to the machine which processes some secret data.

Example:

- **DRM** movie provider Channel+.
- No encryption → The plain movie can be retrieved.

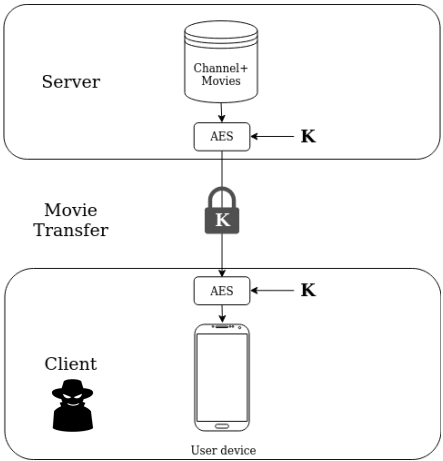


Introduction: The Whitebox model

Whitebox: The attacker has **full access** to the machine which processes some secret data.

Example:

- **DRM** movie provider Channel+.
- No encryption → The plain movie can be retrieved.
- Standard encryption → The key can be read in memory.

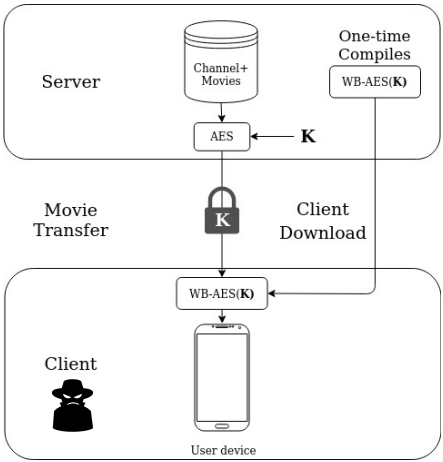


Introduction: The Whitebox model

Whitebox: The attacker has **full access** to the machine which processes some secret data.

Example:

- **DRM** movie provider Channel+.
- No encryption → The plain movie can be retrieved.
- Standard encryption → The key can be read in memory.
- Channel+ needs to hide the key in the client binary.
- Channel+ compiles the client binary **with a hidden key** inside.



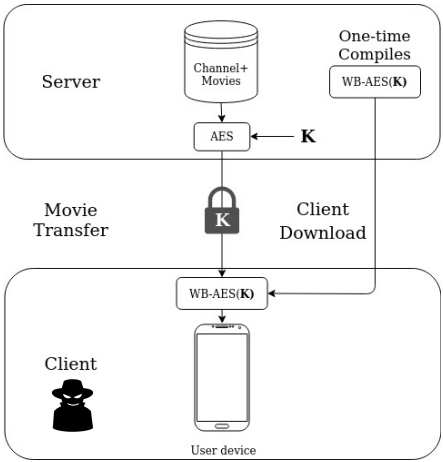
Introduction: The Whitebox model

Whitebox: The attacker has **full access** to the machine which processes some secret data.

Example:

- **DRM** movie provider Channel+.
- No encryption → The plain movie can be retrieved.
- Standard encryption → The key can be read in memory.
- Channel+ needs to hide the key in the client binary.
- Channel+ compiles the client binary **with a hidden key** inside.

No theoretical defense exists in the whitebox model.
In practice, several layers of defense are combined.



A whitebox protection: The data encoding

First attempt of whitebox protection, **the data encoding** on AES[1]:

- **Random bijections are generated** during compilation.
- Inner AES states are split into words and **encoded with the bijections**.
- **Bijections are composed** with the MixColumns, SubBytes and AddKey steps.
- **Not sufficient by itself**[2].

Bottom line

- There are **large key-dependant lookup tables** (≈ **Sboxes**) in the client binary.
- The whitebox decryption is way slower than standard AES.

In this talk, we try to speed up whitebox-compiled code by bitslicing the Sboxes

[1] Chow et al., White-Box Cryptography and an AES Implementation, SAC 2002
[2] Billet et al, Cryptanalysis of a White Box AES Implementation, SAC 2004

The bitslicing technique

Bitslicing

Bitslicing[3] is a **software implementation technique** used to speed-up standard implementations.

[3] Eli Biham, A fast new DES implementation in software, FSE 1997

The bitslicing technique

Bitslicing

Bitslicing[3] is a **software implementation technique** used to speed-up standard implementations.

Bitslicing provides:

- A **performance gain** with bit parallelism.
- A **constant time** implementation by the removal of lookup table accesses.

How?

- Lookup tables are replaced by **logical circuits**.

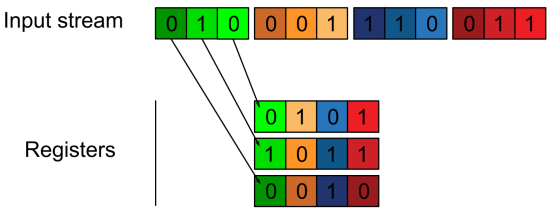
[3] Eli Biham, A fast new DES implementation in software, FSE 1997

A new data representation

Key Idea

- **The data is transposed** before and after processing.

Example: 4 variables of 3 bits

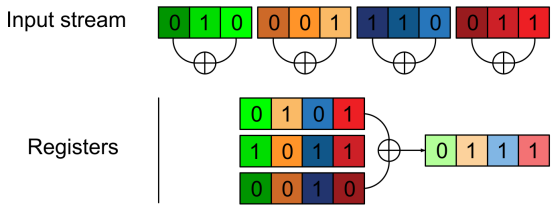


Processing bitsliced data

Bitslicing operations

- **Only bitwise operations** on the registers.
- **Parallelization** of operations.

Example: 4 XORs executed in parallel with a single operation



Bitslicing operations

Operations easy to bitslice:

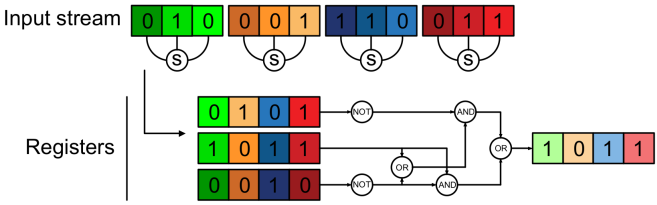
- Bit shuffling ($\ll$, rot...).
- Bitwise operations ($\wedge$, $\vee$, $\oplus$...).
- Modular operations (+,-...).

But **Sboxes** are **hard** to bitslice:

- Use the **circuit representation.**
- Convert it into assembly code.

Difficult to find a light circuit.

abc	000	001	010	011	100	101	110	111
S(abc)	1	0	1	1	0	0	1	0



Circuit representation of a 3-to-1 Sbox S

Bitslicing for Whitebox

Pros:

Performance Gain with parallelization.

Obfuscation Mixed with obfuscation passes.

Cons:

Compilation time Need to bitslice Sboxes.

In this talk, we try to find a **Sbox bitslicing algorithm** that is:

Fast Numerous runs per compilation → 0.1sec to 10min per run.

Efficient Low number of circuit gates, for execution performance.

Generic Sboxes of various sizes need to be bitsliced.

Optimal circuit generation

First idea: Generate **optimal circuits**.

- SAT Solvers solve time is **unpredictable** and **very high** for large Sboxes[4].
- Exhaustive search of assembly instructions already takes days fo 4x4 Sboxes[5].
- More generally, the problem is of **double exponential complexity**[6].

Not practical! We have to generate suboptimal circuits.

[4] Stoffelen, Optimizing S-box Implementations for Several Criteria using SAT Solvers, FSE 2016
[5] Osvik, Speeding Up Serpent, MISC 2000
[6] Buchfuhrer et Umans, The complexity of Boolean formula minimization, JCSS 2011

A big Sbox example: the AES Sbox

- **AES Sbox:** 8x8 → Impossible to generate the optimal circuit.
- **Idea: Bitslice using the structure of the Sbox.**
- **2006:** 132 gates[7].
- **2010:** 115 gates[8].
- **2016:** 113 gates[9].

For whitebox, we suppose that Sboxes have no structure.

[7] Ribeiro et al., Bitslice Implementation of AES, CANS 2006
[8] Boyar et Peralta, Logic Minimization Techniques with Applications to Cryptology, JoC 2013
[9] Calik, http://www.cs.yale.edu/homes/ peralta/CircuitStuff/CMT.html, 2016

Another big Sbox example: DES Sboxes

- DES Sboxes have no particular structure.
- 6x4 Sboxes: Impossible to generate the optimal circuit.
- First circuits generated with **Binary Decision Diagrams** (BDDs)[10].
- Improved BDD algorithm gave the lightest circuit for 13 years[11].

Let's dive into Boolean functions and BDDs!

[10] Eli Biham, A fast new DES implementation in software, FSE 1997

[11] Matthew Kwan,Reducing the Gate Count of Bitslice DES, Cryptology ePrint Archive 2000

Boolean functions

Definition

A Boolean function takes **n** Boolean variables as input and returns a Boolean.

Sboxes and Boolean functions

n-to-**m** Sboxes can be represented by **m** Boolean functions of **n** bits.

Natural way to generate a circuit:

1. Find a Boolean formula for each output bit of a Sbox.
2. Convert them into logical circuits: 1 operation = 1 gate.
3. Optimize (e.g. factorize equivalent sub-circuits).

Canonical forms

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

Disjonctive Normal Form (DNF), sum of products of literals :

$$(\neg \mathbf{a} \wedge \mathbf{b} \wedge \neg \mathbf{c}) \vee (\neg \mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}) \vee (\mathbf{a} \wedge \neg \mathbf{b} \wedge \mathbf{c}) \vee (\mathbf{a} \wedge \mathbf{b} \wedge \neg \mathbf{c}) \vee (\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c})$$

- Can be generated efficiently from the ones in the truthtable.
- Naive average gate count: $1.5 \times n \times 2^{n-1}$
- **Hard to minimize**: the Quine-McCluskey algorithm. $\rightarrow (\mathbf{a} \wedge \mathbf{c}) \vee \mathbf{b}$
  - ▷ Double exponential complexity → **Unreasonable for 8-bit Boolean functions**.
  - ▷ Anyway, **Boolean formula minimization** ≠ **Circuit minimization**.

Canonical forms

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

Conjonctive Normal Form (CNF), product of sums of literals :

$$(\mathbf{a} \vee \mathbf{b}) \wedge (\mathbf{b} \vee \mathbf{c})$$

Equivalent to the **DNF** representation of ¬f(¬x)

Canonical forms

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

Algebraic Normal Form (ANF), XOR of products of variables:

$$\mathbf{b} \oplus (\mathbf{a} \wedge \mathbf{c}) \oplus (\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c})$$

Unique (polynomial) representation:

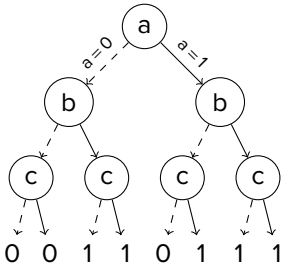
- Easy to generate.
- The **ANF** circuit has in average approx. $n \times 2^{n-1}$ gates.

Binary Decision Diagrams (BDD)s

Decision tree built from a Boolean function:

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

The leaves are the **truthtable values**.

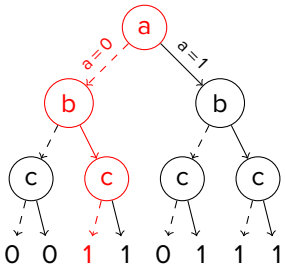


Binary Decision Diagrams (BDD)s

Decision tree built from a Boolean function:

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

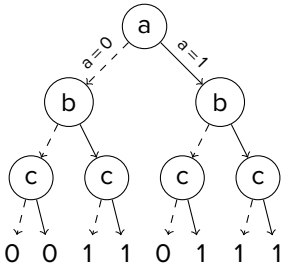
$abc = 010 \rightarrow f(abc) = 1$



Binary Decision Diagrams (BDD)s

Decision tree built from a Boolean function:

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1



A BDD can be represented recursively by logical operations with **4 gates/node**:

$$(\neg a \wedge \mathrm{LT}) \vee (a \wedge \mathrm{RT})$$

where LT and RT are the left and right children (they depend on b, c).

2^{n-1} effective nodes $\rightarrow \approx \mathbf{4} \times \mathbf{2^{n-1}} = \mathbf{2^{n+1}}$ **gates**

First optimization: BDD Transformation

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

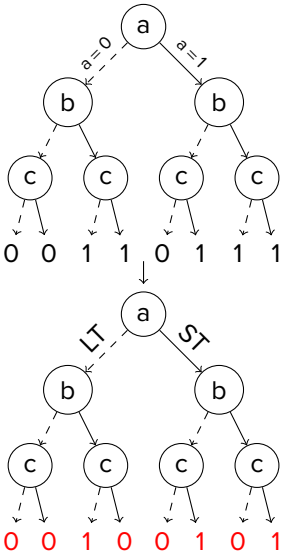
Reduce from 4 to 2 gates per node:

$$(\neg x \wedge LT) \vee (x \wedge RT) \rightarrow (x \wedge ST) \oplus LT$$

With $ST = LT \oplus RT$.

Linear transformation of the leaves.

$2^{n+1} \rightarrow 2^n$ gates.



First optimization: BDD Transformation

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

Reduce from 4 to 2 gates per node:

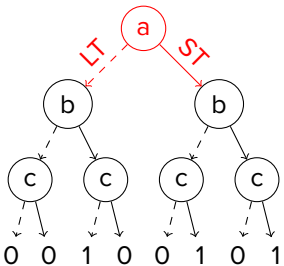
$$(\neg x \wedge LT) \vee (x \wedge RT) \rightarrow (x \wedge ST) \oplus LT$$

With $ST = LT \oplus RT$.

Linear transformation of the leaves.

$2^{n+1} \rightarrow 2^n$ gates.

Example: abc = 101



First optimization: BDD Transformation

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

Reduce from 4 to 2 gates per node:

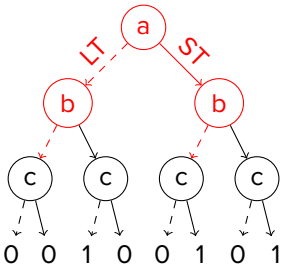
$$(\neg x \wedge LT) \vee (x \wedge RT) \rightarrow (x \wedge ST) \oplus LT$$

With $ST = LT \oplus RT$.

Linear transformation of the leaves.

$2^{n+1} \rightarrow 2^n$ gates.

Example: abc = 101



First optimization: BDD Transformation

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

Reduce from 4 to 2 gates per node:

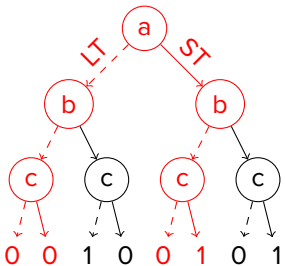
$$(\neg x \wedge LT) \vee (x \wedge RT) \rightarrow (x \wedge ST) \oplus LT$$

With $ST = LT \oplus RT$.

Linear transformation of the leaves.

$2^{n+1} \rightarrow 2^n$ gates.

Example: abc = 101



First optimization: BDD Transformation

abc	000	001	010	011	100	101	110	111
f(abc)	0	0	1	1	0	1	1	1

Reduce from 4 to 2 gates per node:

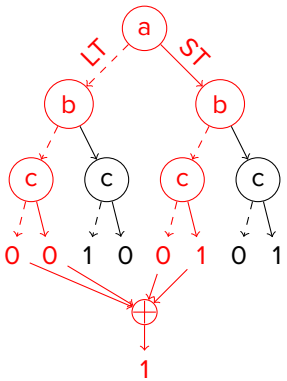
$$(\neg x \wedge LT) \vee (x \wedge RT) \rightarrow (x \wedge ST) \oplus LT$$

With $ST = LT \oplus RT$.

Linear transformation of the leaves.

$2^{n+1} \rightarrow 2^n$ gates.

Example: $abc = 101 \rightarrow f(abc) = 1$



Why are BDDs interesting?

- They are **fast:** $\mathbf{m} \times \mathbf{n} \times 2^{\mathbf{n}}$ operations to generate the circuit of a **n**-to-**m** Sbox.
- They seem **efficient:** $\mathbf{m} \cdot 2^{\mathbf{n}}$ gates (vs. $\mathbf{m} \times \mathbf{n} \times 2^{\mathbf{n-1}}$ for naive ANF).
- They are **generic:** Can be applied to various sizes of Sboxes.

Gate generation process

Boolean functions are interpreted as **truthtables**.

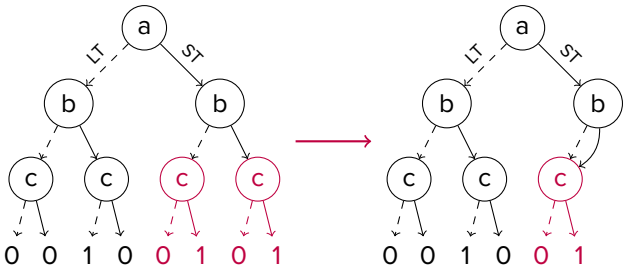
For each output bit of the Sbox, generate the gates **recursively**:

- Given a truthtable **T** (n inputs), chose a selector bit a_i.
- Compute the truthtables **LT** and **ST** ($n-1$ inputs).
- Generate the circuits of **LT** and **ST** recursively.
- Return the circuit $(\mathbf{a_i} \wedge \mathbf{ST}) \oplus \mathbf{LT}$.

Optimizations

We can **reuse subtrees** to reduce the gate count:

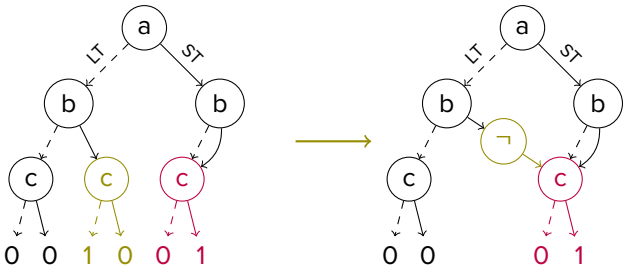
- **Store generated circuits in a hashtable**.
- Use circuits in the hashtable when possible.



Optimizations

We can **reuse subtrees** to reduce the gate count:

- **Store generated circuits in a hashtable**.
- Use circuits in the hashtable when possible.
- Use the **negation** of circuits in the hashtable when possible.



Optimizations

We can **reuse subtrees** to reduce the gate count:

- **Store generated circuits in a hashtable**.
- Use circuits in the hashtable when possible.
- Use the **negation** of circuits in the hashtable when possible.

We may perform a **more advanced** (but costlier) reduction:

- Use the **AND, OR** or **XOR** combinations of already visited circuits when possible.

Optimizations

We can **reuse subtrees** to reduce the gate count:

- **Store generated circuits in a hashtable**.
- Use circuits in the hashtable when possible.
- Use the **negation** of circuits in the hashtable when possible.

We may perform a **more advanced** (but costlier) reduction:

- Use the **AND, OR** or **XOR** combinations of already visited circuits when possible.

Also, the **input bits can be selected in any order**:

- Perform the same algorithm with different input bit selection orders.
- Some trees are optimized better.
- Select the **lightest circuit**.

Results

B : Transformed BDDs with the cheap factorisation.
A : **B** with the advanced reduction.
A_{100} : **A** with 100 input bit reorderings.
N : Naive transformed BDD.

Circuit generation time (milliseconds)†

Sbox size	**B**	**A**	**A_{100}**
4x4	0.2	1.0	11.7
6x4	1.2	11.1	1130
6x6	2.0	19.8	2050
8x8	8.9	508	51500

Circuit size (number of gates)†

Sbox size	**B**	**A**	**A_{100}**	**N**	**DES**
4x4	27	26	22	32	
6x4	105	94	87	128	44
6x6	146	130	119	192	
8x8	633	551	534	1028	

†Average on 100 randomly generated Sboxes.

Comparison with similar work: **DES Sboxes**[12].

[12] Roman Rusakov, https://www.openwall.com/lists/john-users/2011/06/22/1, 2011

Limitations of the approach

Objective: Minimize the execution time on modern CPUs.
We computed the **correlation** between **execution time** and:

The number of gates

Sbox size	Correlation
4x4	0.45
4x6	0.43
6x4	0.38
6x6	0.23
8x8	0.12

The number of register spills

Sbox size	Correlation
4x4	0
4x6	0
6x4	0.07
6x6	0.45
8x8	0.99

Interpretation:

- Too many intermediate values of the circuit → register spills.
- Minimizing the number of gates is questionable for large Sboxes.
- However, hard to know the number register spills from the BDD before compilation.

Conclusion and Open questions

- Bitslicing for whitebox requires a fast, efficient and generic Sbox bitslicer.
- We explored Binary Decision Diagrams which give good results for DES Sboxes.
- Several optimization sets can be chosen for different trade-offs.
- **Still unclear what the best circuit metric**, especially for large Sboxes.