White-box attack resistant cryptography

Hiding cryptographic keys against the powerful attacker

Dušan Klinec, Petr Švenda {xklinec, svenda}@fi.muni.cz
Outline

• CEF&CED, fully homomorphic encryption
• Whitebox cryptography, whitebox AES
• Available implementations & attacks
• Future work, related R&D at CROCS@FIMU
• Garbled circuits [Yao 86]
Troubles with Keys

Protecting key material for cryptographic functions
Standard vs. whitebox attacker model
OllyDbg – key value is static string
OllyDbg – key is visible in memory
What if AES usage is somehow hidden?
Whitebox attacker model

• The attacker is able to:
  – inspect and disassemble binary (static strings, code...)
  – observe/modify all executed instructions (OllyDbg...)  
  – observe/modify used memory (OllyDbg, memory dump...)
• How to still protect value of cryptographic key?
• Who might be whitebox attacker?
  – Mathematician (for fun)
  – Security researcher / Malware analyst (for work)
  – DRM cracker (for fun&profit)
  – ...

Classical obfuscation and its limits

• Time-limited protection
• Obfuscation is mostly based on obscurity
  – add bogus jumps
  – reorder related memory blocks
  – transform code into equivalent one, but less readable
  – pack binary into randomized virtual machine
  – ...
• Barak’s (im)possibility result (2001)
  – family of functions that will always leak some information
  – but practical implementation may exists for others
Computation with Encrypted Data and Encrypted Function

CEF&CED
Scenario

• We’d like to compute function \( F \) over data \( D \)
  – secret algorithm \( F \) or sensitive data \( D \) (or both)
• Solution with trusted environment
  – my trusted PC, trusted server, trusted cloud…
• Problem: can be cloud or client really trusted?
  – server hack, DRM, malware…
• Attacker model
  – controls execution environment (debugging)
  – sees all instructions and data executed
CEF

• Computation with Encrypted Function (CEF)
  – A provides function F in form of P(F)
  – P can be executed on B’s machine with B’s data D as P(D)
  – B will not learn function F during computation
CED

- Computation with Encrypted Data (CED)
  - B provides encrypted data D as $E(D)$ to A
  - A is able to compute its $F$ as $F(E(D))$ to produce $E(F(D))$
  - A will not learn D
CED via homomorphism

1. Convert your function into circuit with additions (xor) and multiplications (and) only
2. Compute addition and/or multiplication “securely”
   – an attacker can compute $E(D1+D2) = E(D1)+E(D2)$
   – but will learn neither $D1$ nor $D2$
3. Execute whole circuit over encrypted data
   – Partial homomorphic scheme
     • either addition or multiplication is possible, but not both
   – Fully homomorphic scheme
     • both addition and multiplication (unlimited)
Partial homomorphic schemes

• Example with RSA (*multiplication*)
  – $E(d1).E(d2) = d1^e \cdot d2^e \mod m = (d1d2)^e \mod m = E(d1d2)$

• Example Goldwasser-Micali (*addition*)
  – $E(d1).E(d2) = x^{d1}r_1^2 \cdot x^{d2}r_2^2 = x^{d1+d2}(r1r2)^2 = E(d1 \oplus d2)$

• Limited to polynomial and rational functions
• Limited to only one type of operation (*mult* or *add*)
  – or one type and very limited number of other type
• Slow – based on modular mult or exponentiation
  – every operation equivalent to whole RSA operation
Fully homomorphic scheme (FHE)

• Holy grail - idea proposed in 1978 (Rivest et al.)
  – both addition and multiplication securely
• But no scheme until 2009 (Gentry)!
  – based on lattices over integers
  – noisy FHE usable only to few operations
  – combined with repair operation
Fully homomorphic scheme - usages

• Outsourced cloud computing and storage (FHE search)
  – Private Database Queries
  – protection of the query content
• Secure voting protocols (yes/no + sum)
• Protection of proprietary info - MRI machines
  – very expensive algorithm analyzing MR data, HW protected
  – central processing restricted due to processing of private patient data
• Read more about current state of FHE
Fully homomorphic scheme - practicality

- Not very practical (yet ^ ) (Gentry, 2009)
  - 2.7GB key & 2h computation for every repair operation
  - repair needed every ~10 multiplication
- FHE-AES implementation (Gentry, 2012)
  - standard PC ⇒ 37 minutes/block (but 256GB RAM)
Protection of cryptographic primitives

Whitebox resistant crypto
White-box attack resistant cryptography

• Problem limited from every cipher to symmetric cryptography cipher only
  – protects used cryptographic key (and data)
• Special implementation fully compatible with standard AES/DES… 2002 (Chow et al.)
  – series of look-ups into pre-computed tables
• Implementation of AES which takes only data
  – key is already embedded inside
  – hard for an attacker to extract embedded key
Advanced Encryption Algorithm

Repeat 10 times

Impractical solution

- Secure, but $2^{128}$ x 16B memory storage
WBACR AES – some techniques

- Pre-compute table for all possible inputs
  - practical for one 16bits or two 8bits arguments table with up to 216 rows (~64KB)
- AddRoundKey: data $\oplus$ key
  - 8bit argument data, key fixed
- Pack several operations together
  - AddRoundKey + SubBytes: $T[i] = S[i] \oplus \text{key}$
- Protect intermediate values by random bijections
  - removed automatically by next lookup
  - $X = F^{-1}(F(X))$
  - $T[i] = S[F^{-1}(i) \oplus \text{key}]$
SubByte(AddRoundKey(x, k))

MixColumn

\[
\begin{bmatrix}
    x_0 \\
    x_1 \\
    x_2 \\
    x_3
\end{bmatrix} = MC \cdot \begin{bmatrix}
    x_0 \\
    0 \\
    0 \\
    0
\end{bmatrix} + MC \cdot \begin{bmatrix}
    0 \\
    x_1 \\
    0 \\
    0
\end{bmatrix} + MC \cdot \begin{bmatrix}
    0 \\
    0 \\
    x_2 \\
    0
\end{bmatrix} + MC \cdot \begin{bmatrix}
    0 \\
    0 \\
    0 \\
    x_3
\end{bmatrix}
\]
SubByte(AddRoundKey(x, k))

Mixing bijection invL prev. rnd

Dual extension

MixColumn +

Mixing Bijection MB 32 x 32

MB⁻¹

Mixing bijection L

IO bijections

state array, round 2

state array, round 3

www.fi.muni.cz/crocs
state array, round 2

IO bijections
Mixing bijection $L_{-1}$ prev. round
Dual extension
SubByte(AddRoundKey(x, k))

MixColumn
+ Mixing Bijection
MB 32 x 32
CB 

Mixing bijection L

IO bijections

state array, round 3

MB \textsuperscript{-1}

\[
\text{SubByte(AddRoundKey(x, k))} + \text{MixColumn} + \text{Mixing Bijection MB} 32 \times 32
\]
Encrypted data

Environment under control
of an attacker

Environment outside control
of an attacker

data

makeTable()

precompTable

AES

key

encrypt(data)

encrypted data

encrypt(data)
Resulting implementation

• More difficult to detect that crypto was used
  – no fixed constants in the code
  – precomputed tables change with every generation
  – even two tables for same key are different
  – (but can still be found)

• Resistant even when precomputed tables are found
  – when debugged, only table lookups are seen
  – key value is never manipulated in plaintext
  – transformation techniques should provide protection to key embedded inside tables
WBACR AES - pros

• Practically usable
  – implementation size ~800KB (tables)
  – speed ~MBs/sec (~6.5MB/s vs. 220MB/s)

• Hard to extract embedded key
  – Complexity semi-formally guaranteed
  – (if the scheme is secure)

• One can simulate asymmetric cryptography!
  – implementation contains only encryption part of AES
  – until attacker extracts key, decryption is not possible
WBACR AES - cons

• Implementation can be used as oracle (black box)
  – attacker can supply inputs and obtain outputs
  – even if she cannot extract the key
  – (can be partially solved by I/O encodings)
• Problem of secure input/output
  – protected is only AES, not code around
• Key is fixed and cannot be easily changed
• Successful cryptanalysis for several schemes
  – several former schemes broken
  – new techniques proposed
List of proposals and attacks

- (2002) First WB AES implementation by Chow et. al. [Chow02]
  - IO bijections, linear mixing bijections, external coding
  - broken by BGE cryptanalysis [Bill04]
    - algebraic attack, recovering symmetric key by modelling round function by system of algebraic equations, $2^{30}$ steps
  - attempt to randomize whitebox primitives, perturbation & random equations added, S-boxes are enc. keys. 4 AES ciphers, major voting for result
  - broken by Mulder et. al. [Mul10]
    - removes perturbations and random equations, attacking on final round removing perturbations, structural decomposition. $2^{17}$ steps
  - broken by Mulder et. al. [Mul12]
    - linear equivalence algorithm used (backward AES-128 compatibility => linear protection has to be inverted in next round), $2^{32}$ steps
- (2011) Protecting white-box AES with dual ciphers [Kar11]
  - broken by our work [Kli13]
    - protection shown to be ineffective
Whitebox transform IS used in the wild

• Proprietary DRM systems
  – details are usually not published
  – AES-based functions, keyed hash functions, RSA, ECC...
  – interconnection with surrounding code
• Chow at al. (2002) proposal made at Cloakware
  – firmware protection solution
• Apple’s FairPlay & Brahms attack
• TrojanSpy:Win32/WhiteBox? :)
• ...

...
Available practical implementations

Demo
Demo – WAES

• WAES tables generator
  – configuration options
  – *.h files with pre-computed tables

• WAES cipher implementation
  – compile-in tables
  – tables as memory blob
## WAES performance

- Intel Core i5 M560@2.67GHz

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
<th>Additional info.</th>
<th>OpenSSL result</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate WB AES</td>
<td>8.48 s avg.</td>
<td>100 samples</td>
<td></td>
</tr>
<tr>
<td>throughput, 1 MB random</td>
<td>867.8 KB/s</td>
<td>1.18 s</td>
<td>57283 KB/s</td>
</tr>
<tr>
<td>throughput, 10 MB random</td>
<td>1022.977 KB/s</td>
<td>10.01 s</td>
<td>54179 KB/s</td>
</tr>
<tr>
<td>throughput, 100 MB random</td>
<td>1028.319 KB/s</td>
<td>99.58 s</td>
<td>74744 KB/s</td>
</tr>
<tr>
<td>throughput, 1024 MB random</td>
<td>1124.792 KB/s</td>
<td>932.24 s</td>
<td>63723 KB/s</td>
</tr>
<tr>
<td>throughput, 1 MB null</td>
<td>975 KB/s</td>
<td>1.05 s</td>
<td>93091 KB/s</td>
</tr>
<tr>
<td>throughput, 10 MB null</td>
<td>969.970 KB/s</td>
<td>10.56 s</td>
<td>68821 KB/s</td>
</tr>
<tr>
<td>throughput, 100 MB null</td>
<td>1058.507 KB/s</td>
<td>96.74 s</td>
<td>56356 KB/s</td>
</tr>
<tr>
<td>throughput, 1024 MB null</td>
<td>1050.593 KB/s</td>
<td>998.08 s</td>
<td>57283 KB/s</td>
</tr>
</tbody>
</table>

Table 4.2: Results of the benchmark for whitebox AES generator
BGE attack in progress

recoverQj; q = 0x88; gamma=0x01;
recoverQ self-test; r=5; col=3; (y0, y3); P[0].deltaInv=0x03; alfa_{3,0}=0x03
recoverQ self-test; r=5; col=3; (y0, y3); P[1].deltaInv=0x01; alfa_{3,1}=0x01
recoverQ self-test; r=5; col=3; (y0, y3); P[2].deltaInv=0x01; alfa_{3,2}=0x01
recoverQ self-test; r=5; col=3; (y0, y3); P[3].deltaInv=0x02; alfa_{3,3}=0x02
recoverQj; q = 0x3c; gamma=0x01;

Going to reconstruct encryption key from extracted round keys...
* Round keys extracted from the process, r=3
  0x3d 0x47 0x1e 0x6d 0x80 0x16 0x23 0x7a 0x47 0xfe 0x7e 0x88 0x7d 0x3e 0x44 0x3b

* Round keys extracted from the process, r=4
  0xef 0xa8 0xb6 0xdb 0x44 0x52 0x71 0x0b 0xa5 0x5b 0x25 0xad 0x41 0x7f 0x3b 0x00

* Round keys extracted from the process, r=5
  0xd4 0x7c 0xca 0x11 0xd1 0x83 0xf2 0xf9 0xc6 0x9d 0xb8 0x15 0xf8 0x87 0xbc 0xbc

Recovering cipher key from round keys...
We have correct Rcon! rconIdx=3
RC=2; previousKey:
  0xf2 0x7a 0x59 0x73
  0xc2 0x96 0x35 0x59
  0x95 0xb9 0x80 0xf6
  0xf2 0x43 0x7a 0x7f

RC=1; previousKey:
  0xa0 0x88 0x23 0x2a
  0xfa 0x54 0xa3 0x6c
  0xfe 0x2c 0x39 0x7e
  0x17 0xb1 0x39 0x05

RC=0; previousKey:
  0x2b 0x28 0x20 0x09
  0x7e 0xae 0xf7 0xcf
  0x15 0xd2 0x15 0x4f
  0x16 0xa6 0x88 0x3c

Final result:
  0x2b 0x7e 0x15 0x16 0x28 0xae 0xd2 0xa6 0xab 0xf7 0x15 0x88 0x09 0xcf 0x4f 0x3c

Benchmark finished! Total time = 3a s; on average = 58 s; clocktime=57.66 s;
What's in our pipeline?

future work
Webpage with implemented proposals

• Obvious next step ♤
• Relevant academic papers didn’t come with implementation
  – true both for proposals and attacks
• Our work provided 2 implementations & 2 attacks
  – we will do remaining soon
• Relevant links
• CrackMe challenges
• http://www.fi.muni.cz/~xsvenda/whiteboxcrypto/
Modifications to W-AES

1. Hash-chain generated round keys
   - Noninvertible

2. Key-dependent confusion / S-boxes
   - High variability (13 bytes dependence)

3. Key-dependent diffusion
   - 32x32 -> 128x128 matrix

4. Incorporating of algebraic incompatible operations
   - Like in IDEA cipher
Summary

• Computation with encrypted data & function
  – strong whitebox attacker model
• Whitebox cryptography tries to be better than classical obfuscation alone
  – mathematical-level proofs for cryptographic primitives
• Implementation of selected schemes (almost \(^\star\) ) released
  – published attacks as well

Questions?
Garbled circuits
Motivation

Borrowed from Vitaly Shmatikov's presentation from CS 380S course

- General framework for describing computation between parties who do not trust each other

Example: elections
- N parties, each one has a “Yes” or “No” vote
- Goal: determine whether the majority voted “Yes”, but no voter should learn how other people voted

Example: auctions
- Each bidder makes an offer
  - Offer should be committing! (can’t change it later)
- Goal: determine whose offer won without revealing losing offers
• Example: distributed data mining
  • Two companies want to compare their datasets without revealing them
  • For example, compute the intersection of two lists of names

• Example: database privacy
  • Evaluate a query on the database without revealing the query to the database owner
  • Evaluate a statistical query on the database without revealing the values of individual entries
  • DNA tests for genetic diseases (Smith-Waterman alg.)
  • Many variations
Example: **homomorphic operation over data**
- Alice provides 2 ciphertexts
- Bob provides encryption key
- Circuit performs decryption, desired operation on plaintext, encryption back

Example: **side channel protection, one time programs**
- One time program consist of:
  - One time memory – one time oblivious transfer
  - Yao circuit
- Very important for smart cards
Yao’s Protocol

- Compute any function securely
  - ... in the semi-honest model
- First, convert the function into a boolean circuit
Multiplexer

Inverters

“AND” gate

“OR” gate

Q
If branches

- Has to evaluate both branches (to leak no information about inputs)
- Then use multiplexer to select the right one

**Original code**

```c
x = read_input();
if (x > 5) {
    y = 7;
} else {
    y = 12;
}
```

**GC compatible**

```c
x = read_input();
c1 = x > 5;
y1 = 7;
c2 = !c1;
y2 = 12;
y = (y1 & c1) || (y2 & c2);
```

multiplexer
Loop unrolling

- Loop condition must be independent on user input
- Unroll loops to depth k, using if conditions

Original code

```c
x = read_input();
y = 0;
for (i=0; i<3; i++) {
    y++;
}
```

GC compatible

```c
x = read_input();
y = 0; i = 0;
if (i<3){
    y1=y+1;
    i1=i+1;
    if (i1<3){
        y2=y1+1;
        i2=i1+1;
        if (i2<3){
            y3=y2+1;
            i3=i2+1;
        }
    }
}
```
Addition

Half adder
Sum = A \oplus B
Carry = A \& B

Full adder
Sum = (A \oplus B) \oplus C_{in}
Carry = ((A \oplus B) \& C_{in})|(A \& B)
1: Pick Random Keys For Each Wire

- Next, evaluate **one gate** securely
  - Later, generalize to the entire circuit
- Alice picks two **random keys** for each wire
  - One key corresponds to “0”, the other to “1”
  - 6 keys in total for a gate with 2 input wires

---

*Borrowed from Vitaly Shmatikov's presentation from CS 380S course*
2: Encrypt Truth Table

- Alice encrypts each row of the truth table by encrypting the output-wire key with the corresponding pair of input-wire keys.

Original truth table:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Encrypted truth table:

- $E_{k_0x}(E_{k_{0y}}(k_{0z}))$
- $E_{k_0x}(E_{k_{1y}}(k_{0z}))$
- $E_{k_{1x}}(E_{k_{0y}}(k_{0z}))$
- $E_{k_{1x}}(E_{k_{1y}}(k_{1z}))$

Borrowed from Vitaly Shmatikov's presentation from CS 380S course
3: Send Garbled Truth Table

- Alice randomly permutes ("garbles") encrypted truth table and sends it to Bob

Garbled truth table:

- $E_{k_0x}(E_{k_0y}(k_{0z}))$
- $E_{k_0x}(E_{k_1y}(k_{0z}))$
- $E_{k_1x}(E_{k_0y}(k_{0z}))$
- $E_{k_1x}(E_{k_1y}(k_{1z}))$
- $E_{k_0x}(E_{k_0y}(k_{0z}))$
- $E_{k_0x}(E_{k_1y}(k_{0z}))$
- $E_{k_1x}(E_{k_1y}(k_{1z}))$
- $E_{k_0x}(E_{k_0y}(k_{0z}))$

Does not know which row of garbled table corresponds to which row of original table
4: Send Keys For Alice’s Inputs

- Alice sends the key corresponding to her input bit
  - Keys are random, so Bob does not learn what this bit is

Learns $K_{b'x}$ where $b'$ is Alice’s input bit, but not $b'$ (why?)

Garbled truth table:

$E_{k_{1x}}(E_{k_{0y}}(k_{0z}))$
$E_{k_{0x}}(E_{k_{1y}}(k_{0z}))$
$E_{k_{1x}}(E_{k_{1y}}(k_{1z}))$
$E_{k_{0x}}(E_{k_{0y}}(k_{0z}))$

If Alice’s bit is 1, she simply sends $k_{1x}$ to Bob; if 0, she sends $k_{0x}$
Oblivious Transfer (OT)

- Fundamental SMC primitive

- A inputs two bits, B inputs the index of one of A’s bits

- B learns his chosen bit, A learns nothing
  - A does not learn which bit B has chosen; B does not learn the value of the bit that he did not choose

- Generalizes to bitstrings, M instead of 2, etc.

Borrowed from Vitaly Shmatikov’s presentation from CS 380S course
5: Use OT on Keys for Bob’s Input

- Alice and Bob run oblivious transfer protocol
  - Alice’s input is the two keys corresponding to Bob’s wire
  - Bob’s input into OT is simply his 1-bit input on that wire

Run oblivious transfer
- Alice’s input: $k_{0y}, k_{1y}$
- Bob’s input: his bit $b$
- Bob learns $k_{by}$
- What does Alice learn?

Garbled truth table:
- $E_{k_{0x}}(E_{k_{0y}}(k_{0z}))$
- $E_{k_{0x}}(E_{k_{1y}}(k_{0z}))$
- $E_{k_{1x}}(E_{k_{0y}}(k_{1z}))$
- $E_{k_{0x}}(E_{k_{1y}}(k_{1z}))$

Borrowed from Vitaly Shmatikov’s presentation from CS 380S course
6: Evaluate Garbled Gate

- Using the two keys that he learned, Bob decrypts exactly one of the output-wire keys
  - Bob does not learn if this key corresponds to 0 or 1
- Why is this important?

Garbled truth table:

- Suppose \( b' = 0, b = 1 \)
- This is the only row Bob can decrypt.
- He learns \( K_{0z} \)

Knows \( K_{b'x} \) where \( b' \) is Alice's input bit and \( K_{by} \) where \( b \) is his own input bit

Borrowed from Vitaly Shmatikov's presentation from CS 380S course
7: Evaluate Entire Circuit

- In this way, Bob evaluates entire garbled circuit
  - For each wire in the circuit, Bob learns only one key
  - It corresponds to 0 or 1 (Bob does not know which)
    - Therefore, Bob does not learn intermediate values (why?)

- Bob tells Alice the key for the final output wire and she tells him if it corresponds to 0 or 1
  - Bob does not tell her intermediate wire keys (why?)
Brief Discussion of Yao’s Protocol

• Function must be converted into a circuit
  – For many functions, circuit will be huge

• If m gates in the circuit and n inputs, then need 4m encryptions and n oblivious transfers
  – Oblivious transfers for all inputs can be done in parallel

• Yao’s construction gives a constant-round protocol for secure computation of any function in the semi-honest model
  – Number of rounds does not depend on the number of inputs or the size of the circuit!

Borrowed from Vitaly Shmatikov’s presentation from CS 380S course
• Secure function evaluation – hidden functions

• What if Bob wants to evaluate a secret function over Alice's input?
  • Credit report check

• 2-party protocols: assume both parties know the function

• Use a concept of an universal circuit (interpreter)
  • \( UC(C, x, y) = C(x, y) \)
  • \( C \) is a input
  • \( UC \) is known by both parties, no information leak
• How to generate a circuit

• Fairplay compiler
  • Specialized language, SFDL, SHDL
• Compile ANSI C to garbled circuit
• (PCF) Portable Circuit Format
  • Byte-code translator, on-the-fly circuit generator
## Latest results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Gates</th>
<th>Non-XOR</th>
<th>Compile</th>
<th>Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128</td>
<td>31 512</td>
<td>44 %</td>
<td></td>
<td>0.2</td>
</tr>
<tr>
<td>Fingerprint match, closest thr.</td>
<td></td>
<td></td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>Face recognition, 900bit HW</td>
<td></td>
<td></td>
<td></td>
<td>0.05</td>
</tr>
<tr>
<td>Smith-Waterman, 60x60</td>
<td></td>
<td></td>
<td></td>
<td>477</td>
</tr>
<tr>
<td>16 384-bit Comparison</td>
<td>97 733</td>
<td>33 %</td>
<td>3.4</td>
<td>4.40</td>
</tr>
<tr>
<td>1024-bit Summation</td>
<td>11 999</td>
<td>25 %</td>
<td>4.60</td>
<td>0.250</td>
</tr>
<tr>
<td>1024-bit Multiplication</td>
<td>25 592 368</td>
<td>25 %</td>
<td>74</td>
<td>40.9</td>
</tr>
<tr>
<td>256-bit RSA</td>
<td>673 105 990</td>
<td>35 %</td>
<td>381</td>
<td>980</td>
</tr>
<tr>
<td>512-bit RSA</td>
<td>5 397 821 470</td>
<td>36 %</td>
<td>350</td>
<td>7300</td>
</tr>
<tr>
<td>1024-bit RSA</td>
<td>42 151 698 718</td>
<td>36 %</td>
<td>564</td>
<td>56 000</td>
</tr>
<tr>
<td>16x16 Matrix mult.</td>
<td>14 303 864</td>
<td>30 %</td>
<td>109</td>
<td>23.7</td>
</tr>
<tr>
<td>Interpreter, 50 gates</td>
<td>1 122 351</td>
<td>33 %</td>
<td>1.15</td>
<td>6.267</td>
</tr>
</tbody>
</table>
Acknowledgments

This work was supported by the project VG20102014031, programme BV II/2 – VS of the Ministry of the interior of the Czech Republic.
Thank you for your attention!

Questions