How to Attack the IoT with Hardware Trojans

hardwear.io
Den Haag, September 22, 2017

Christof Paar
Ruhr Universität Bochum & University of Massachusetts Amherst
Acknowledgement

- Georg Becker
- Pawel Swierczynski
- Marc Fyrbiak
Agenda

- Introduction to Hardware Trojans
- Sub-Transistor ASIC Trojans
- FPGA Trojan
- Key extraction attack
- Auxiliary Stuff
Agenda

- Introduction to Hardware Trojans
- Sub-Transistor ASIC Trojans
- FPGA Trojan
- Key extraction attack
- Auxiliary Stuff
Hardware Trojans

Malicious change or addition to an IC that adds or remove functionality, or reduces reliability

Many rather unpleasant “applications”
Trojan Injection & Adversaries Scenarios

DoD scenario 2005

- Manufacturing
  Malicious factory, esp. off-shore (foreign Government)

- Design Manipulation
  - 3rd party IP-cores
  - malicious employee

not-so-unlikely 2013

- During shipment
  cf. NSA’s *interdiction*

- Built-in
  backdoors etc.
Where are we with “real” HW Trojans?

- No true hardware Trojan observed in the wild
- All examples from academia
- Vast majority of publications focus on detection
Our Thoughts

1. Designing Trojan could be fun too
2. Especially those that go undetected
Simple Example: Inverter Trojan

Let’s modify an inverter so that it always outputs “1” (VDD) without visible changes.

\[
\begin{array}{c|c}
A & Y \\
0 & 1 \\
1 & 0 \\
\end{array}
\]

\[\text{AY} \]

\[\text{VDD} \]

\[\text{GND} \]

\[\text{VDD} \]

\[\text{GND} \]
PMOS Transistor Trojan

Unmodified PMOS transistor

Trojan trans. w/ constant VDD output
“Always One” Trojan Inverter

<table>
<thead>
<tr>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Q1: Can the manipulation be detected?
Q2: How to build a **useful** Trojan from here?
Detection: layout view of Trojan inverter

Which one has the Trojan?

Original Inverter  “Always One” Trojan

Unchanged:
• All metal layers
• Polysilicon layer
• Active area
• Wells

⇒ Dopant changes (very ?) difficult to detect using optical inspection!
“Small” remaining question

- Unfortunately, circuits will not function correctly with this simple stuck-at fault...
- ...functional testing (after manufacturing) will detect fault right away

Q2: Can we build a **meaningful** Trojan using dopant modifications that passes functional testing?
A Real-World True Random Number Generator

... random numbers generate cryptographic keys for

• secure web browsing
• email encryption
• document certification
• ...

Disclaimer: Attacks work against most modern TRNGs

dopant Trojan
2 Modules form Random Number Generator

- Entropy source
- Digital post processing

128

Crypto Key

011001011110 ...

Intel Core™i7
Inside the Random Number Generator

- **State register k**
  - 0 0 1 1 0 1 0 1 1 1 ... 0

- **State register c**
  - 1 0 0 1 0 0 0 1 1 0 ... 1

- **AES**

- **Crypto Key**

- **011001011110 ...**

- **256 random bits**

- **1,000,000,000,000,000,000,000,000,000,000,000,000,000 possible crypto keys**

- **testing all keys: lifetime of the universe**
Trojan Random Number Generator

224 Trojan bits (fixed by attacker!)

\[ \begin{array}{cccccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
\end{array} \ldots \ 1 \]

\[ \begin{array}{cccccccccc}
c_1 & c_2 & \ldots & c_{32} & 0 & 0 & 1 & \ldots & 0 \\
\end{array} \]

only 32 random bits

\[ \begin{array}{ccccccc}
128 & & & & & & 128 \\
\end{array} \]

\[ +1 \]

\[ \begin{array}{ccccccc}
128 & & & & & & 128 \\
\end{array} \]

\[ \begin{array}{ccccccc}
128 & & & & & & 128 \\
\end{array} \]

AES

128

Crypto key

\[ \begin{array}{ccccccc}
128 & & & & & & 128 \\
\end{array} \]

\[ \begin{array}{ccccccc}
128 & & & & & & 128 \\
\end{array} \]

\[ \begin{array}{ccccccc}
128 & & & & & & 128 \\
\end{array} \]

-testing all keys: few seconds

-1,000,000,000,000,000,000,000,000,000,000,000,000,000 possible crypto keys

... but circuit would still be tested as “faulty” during manufacturing...
Built-in self test prevents detection of fault

- **256 bit state Rate Matcher (Based on AES)**
- **Test Mode**
  - Known input
  - 512 bits
  - CRC Checksum 32 bits
  - Reference Checksum
    - ≠ Due to clever choosing of the Trojan bits

- **TROJAN Rate Matcher (Based on AES)**
  - Known input
  - 512 bits
  - CRC Checksum 32 bits
  - Reference Checksum
    - ≠ Due to clever choosing of the Trojan bits
Meaningful hardware Trojans are possible without extra logic
Many detection techniques don’t guarantee a Trojan free design!
Built-in self tests can be dangerous
More details:
Becker, Regazzoni, P, Burleson, *Stealthy Dopant-Level Hardware Trojan*. CHES 2013

... but the scientific community functions as it is supposed to do:

Trojan detection is possible w/ scanning electron microscope
Sugawara et al., *Reversing Stealthy Dopant-Level Circuits*. CHES 2014
Agenda

- Introduction to Hardware Trojans
- Sub-Transistor ASIC Trojans
- FPGA Trojan
- Key extraction attack
- Auxiliary Stuff
FPGAs = Reconfigurable Hardware
... are widely used

world market:
≈ 5b devices
Can an we build **hardware** Trojans by manipulating the bitstream?

Configuration file “bitstream”
Principle of FPGA-based Trojans

- Manipulate Bits
  - small look-up tables realize logic

configure

Source Graphics: SimpleIcon, Xilinx
The Mechanics of FPGAs

Two challenges
1. find AES in unknown design
2. meaningful manipulation

10^3 ... 10^6 logic cells

bitstream is complex and proprietary
Finding AES:
Luckily, crypto has very specific components

- S-boxes are realized as 6x1 look-up tables (LUTs)
- LUT locations can be found in bitstream
- S-box contents is very specific (luckily)
AES detection in practice

8 different real-world AES implementations

<table>
<thead>
<tr>
<th>Impl.</th>
<th>Architecture</th>
<th>AES</th>
<th>LUTs with S-box logic</th>
<th>S-boxes in memory</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Round-based</td>
<td>128</td>
<td>$(16+4) \cdot 32 = 640$</td>
<td>no</td>
<td>100 %</td>
</tr>
<tr>
<td>#2</td>
<td>$\frac{1}{4}$ Round</td>
<td>128</td>
<td>0</td>
<td>yes</td>
<td>100 %</td>
</tr>
<tr>
<td>#3</td>
<td>$\frac{1}{4}$ Round</td>
<td>192</td>
<td>0</td>
<td>yes</td>
<td>100 %</td>
</tr>
<tr>
<td>#4</td>
<td>$\frac{1}{4}$ Round</td>
<td>256</td>
<td>0</td>
<td>yes</td>
<td>100 %</td>
</tr>
<tr>
<td>#5</td>
<td>Round-based</td>
<td>128</td>
<td>$(0+4) \cdot 32 = 128$</td>
<td>yes</td>
<td>100 %</td>
</tr>
<tr>
<td>#6</td>
<td>Round-based</td>
<td>128</td>
<td>0</td>
<td>yes</td>
<td>100 %</td>
</tr>
<tr>
<td>#7</td>
<td>Round-based</td>
<td>128</td>
<td>0</td>
<td>yes</td>
<td>100 %</td>
</tr>
<tr>
<td>#8</td>
<td>Round-based</td>
<td>128</td>
<td>$(16+4) \cdot 32 = 640$</td>
<td>no</td>
<td>100 %</td>
</tr>
</tbody>
</table>

TABLE IV: Overview of evaluated AES implementations
Algorithm substitution attack and its implications

1. Inject weak S-boxes in bitstream

2. Trojan AES is configured

‘Useful’ attacks are still possible!

1. Storage encryption – Plaintext recovery
   • Attacker can recover plaintext without access to $k$

2. Temporary device access – Key extraction
   • switch S-box and recover $k$ from $CT$
   • configure original S-box
Conclusion

- New attack vector against FPGAs!
- Reconfigurability allows “hardware” Trojans designed in the lab
- Bitstream protection is crucial!
  (but not easy, cf. our work at CCS 2011 & FPGA 2013)
- Details at:
Agenda

- Introduction to Hardware Trojans
- Sub-Transistor ASIC Trojans
- FPGA Trojan
- Key extraction attack
- Auxiliary Stuff
What else can we do with bitstream manipulations?

Hmm, are there simpler ways to extract keys through bitstreams without Trojans?
Set-Up

Can bitstream manipulation of unknown design lead to key leakage?

classical known-plaintext set-up

non-classical set-up: alteration of bitstream

\[ PT \quad \text{CT} = \text{AES}(k, \text{PT}) \]

??
**Bitstream Fault Injections (BiFI)**

/ Image of a circuit board with an AES inside label and a key symbol.

---

$PT \quad CT = AES (k, PT)$

**(surprising) attack strategy**

1. manipulate 1st LUT table (e.g., all-zero)
2. configure FPGA
3. send PT
4. check: Does CT contain $k$?  
   if not: GOTO 1 and manipulate next LUT

---

10-30k LUTs per FPGA

---
How exactly does the key leak??

Many leakage hypotheses
- CT = roundkey
- CT = inverted roundkey
- CT = PT xor roundkey
- ...

Many LUT manipulations possible
- all-zero
- all-one
- invert
- upper half of LUT all-zero
- ...

\[ PT \rightarrow CT = \text{AES}(k, PT) \rightarrow \text{configure} \]

\[ 100101010101010101010100 \]
\[ 0011101001011101100000 \]
\[ 0001010111010100110011 \]
\[ 1010110001100101011111 \]

[Diagram of AES inside and LUT manipulations]
Results for Bitstream Fault Injections (BiFI)

Real world attack
- 16 unknown AES designs (Internet)
- 16 different manipulation rules
- $\approx 20k$ LUTs
- 3.3 sec for configuring and checking one alterations

Results
- successful key extraction for every design!
- on average $\approx 2000$ configurations ($\approx 2h$)
- works even for encrypted bitstream (w/o MAC)
Conclusion

- Bitstream Fault Injections (BiFI) is a new family of fault attacks
- Malleability of bitstream is major weakness for FPGAs!
- Are there more bitstream-based attacks?
- Details at:
Agenda

- Introduction to Hardware Trojans
- Sub-Transistor ASIC Trojans
- FPGA Trojan
- Key extraction attack
- Auxiliary Stuff
Related Workshops

CHES – Cryptographic Hardware & Embedded Systems
25.-28. September 2017, Taiwan

escarEurope – Embedded Security in Cars
Berlin, November 2017
Easy-to-understand book for applied cryptography

Introduction to Cryptography by Christof Paar

24 video lectures
Thank you very much for your attention!

Christof Paar

Ruhr-Universität Bochum