EC 500
Hardware Security

Notes on Hardware Security Primitives

Prof. Michel A. Kinsky

Department of Electrical & Computer Engineering
Cryptography vs. System Security

- Encryption

\[
\text{Plaintext (m)} \xrightarrow{\text{Encryption}} \text{Ciphertext (c)} \xleftarrow{\text{Decryption}} \text{Plaintext (m)}
\]

\[
\text{secret key (k)} \rightarrow \text{Encryption (Enc)} \rightarrow \text{Ciphertext (c)} \rightarrow \text{Decryption (Dec)} \rightarrow \text{Plaintext (m)}
\]

- Decryption is the inverse function

• But where are these secret keys stored?
Cryptography vs. System Security

- One of the challenges is the security of the secret key itself
- Storing digital information, like an encryption key, in a device in a way that is resistant to physical attacks is very difficult and expensive
Hardware Security Primitive

- They help to anchor the security of the system in the hardware layer
  - Hardware as the root of the trust

- Examples
  - Circuit Level
    - Hardware obfuscation
  - Digital Design
    - IC watermarking
  - Datapath & Control
    - Self-repair and regeneration of datapaths
  - Component Level primitives
    - PUF, ORAM, RNG, ...
Physical Unclonable Functions (PUF)

- Physical Unclonable Functions (PUFs) have been introduced as the hardware equivalent of a one-way function
  - Due to random process variations, no two Integrated Circuits even with the same layouts are identical
  - Variation is inherent in fabrication process
    - Even circuits produced by the same design and technology will have slight difference/variations
  - Hard to remove or predict
    - Unpredictable
      - To users
      - To manufacturers (even the manufacturer cannot produce two identical PUFs)
    - Unclonable
      - For the most part

- A PUF can be used as an unclonable key
Physical Unclonable Functions (PUF)

Applications:

• Secret Key Generation / Storage
• Random Number Generator
• Identification
• Authentication
• Hardware Obfuscation
• Key exchange
• …
Physical Unclonable Functions (PUF)

- Properties:
  - Work Principle: Challenge-Response Pairs (CRPs)
    - Before PUF deployment
      - Challenges
      - Responses
      - CRPs Storage
    - At authentication
      - Challenge $i$
      - Response $i$
      - Matches Record?
      - Response $i$
Physical Unclonable Functions (PUF)

- Computable
  - Given PUF and $x$, it is easy to evaluate $y = PUF(x)$

- Unique
  - $PUF(x)$ contains some information about the identity of the physical entity embedding the PUF

\[ \begin{align*}
  &\text{Challenge } C \\
  &\text{PUF}_1 \rightarrow \text{Response } R_1 \\
  &\text{PUF}_2 \rightarrow \text{Response } R_2 \\
  &\text{With } R_1 \neq R_2
\end{align*} \]
Physical Unclonable Functions (PUF)

- **Computable**
  - Given PUF and x, it is easy to evaluate $y = PUF(x)$

- **Unique**
  - $PUF(x)$ contains some information about the identity of the physical entity embedding the PUF

- **Reproducible**
  - $y \approx PUF(x)$ is reproducible - up to a small error

- **Unclonable**
  - Given PUF, it is hard to construct a procedure $PUF'$ where $PUF(x) \approx PUF'(x)$

- **Unpredictable**
  - Given a set of CRPs, it is hard to predict $y \approx PUF(x)$
  - Meaning learning is hard

- **One-way**
  - Given only $y$ and the corresponding PUF, it is hard to find $x$ such that $y \approx PUF(x)$
Physical Unclonable Functions (PUF)

- There are more types of PUF implementations

PUF
  ▪ Silicon PUF
    - Glitch PUF
    - Memory PUF
    - Delay PUF
    ▪ Arbiter PUF
    ▪ Ring O. PUF
  ▪ Non-silicon PUF
    - Optical PUF
    - Acoustic PUF
    ▪ Coating PUF
Physical Unclonable Functions (PUF)

- By the source of randomness
  - PUFs Using Explicitly-introduced Randomness
  - *Easier to control PUF uniqueness*
    - Optical PUF
    - Coating PUF
  - PUFs Using Intrinsic Randomness
  - *More popular, no modification to the original design*
    - Delay PUF – ring oscillator, arbiter PUFs etc.
    - Memory PUF – SRAM, DRAM, FF PUFs etc.
    - Mixed signal PUF – analog PUFs
    - Other types – Bi-stable Ring, magnetic stripe card, quantum confinement PUF etc.
Physical Unclonable Functions (PUF)

- **By the Size of Challenge-Response Pairs (CRPs)**
  - **Weak PUFs**
    - Small size of CRP set (usually 1)
    - Mostly used for key storage
    - The CRP access has to be restricted from attackers
  - **Strong PUFs**
    - Large size of CRP set
    - Mostly used for authentication
    - A portion of CRP set can be public
    - Impossible to predict the unknown CRPs
Popular PUF Designs

- **SRAM PUF - A weak memory PUF**
  - Memory cell (a cross-coupled inverter) based
  - Using the intrinsic randomness in each cell’s initial state at power up
  - Easy to implement, but not applicable to all FPGAs
    - Some modern FPGAs assign fixed value to the cells’ initial state
Popular PUF Designs

- **Arbiter PUF**– A strong delay PUF
  - MUX based
  - Using the intrinsic delay differences in each MUX
  - Stronger PUF
    - n challenges produce $2^n$ possible routes (responses)
  - Hard to implement on FPGAs (explained later)
Popular PUF Designs

- **Ring Oscillator (RO) PUF – A Delay PUF**
  - Using the intrinsic delay differences in each inverter (LUT);
  - Weak PUF (**not that weak**)
    - $\frac{n(n-1)}{2}$, where $n = \# \text{ of ROs per RO group}$
  - Easier to implement on FPGA but costs more area than Arbiter PUF
Popular PUF Designs

- **Ring Oscillator (RO) PUF – A Delay PUF**
  - Using the intrinsic delay differences in each inverter (LUT);
  - Weak PUF (*not that weak*)
    - $\frac{n(n-1)}{2}$, where $n = \# \text{ of ROs per RO group}$
  - Easier to implement on FPGA but costs more area than Arbiter PUF
Key Points in PUF Designs

- Memory PUFs
  - Bit selection
    - Although the initial value of all cells at start up is unpredictable
    - The stable ones should be selected for the PUF response
    - A stable cell: it is read as 1 or 0 at most boot-ups

- Black: stable 1 - White: stable 0
- Grey: unstable bits - should not be selected
Key Points in PUF Designs

- **Delay PUFs**
  - **Symmetric place & route**
    - The two racing routes need to be identical / symmetric
    - So that the only factor determining the delay difference is each cell’s manufacturer variation, instead of the routing difference.
Key Points in PUF Designs

- Delay PUFs
  - Symmetric place & route
    - The two racing routes need to be identical / symmetric
    - So that the only factor determining the delay difference is each cell’s manufacturer variation, instead of the routing difference.
Fuzzy Extractor and Helper Data

- Enrollment process
Fuzzy Extractor and Helper Data

- Reconstruction process
Attacks on PUF: Clone the Unclonable

- Exhaustive Reading of the Weak PUFs
  - Reading out the only 1 CRP on memory PUFs On chip channel

- Modeling the Strong PUFs
  - With the large public subset of the CRPs of Arbiter, RO PUFs.
  - Machine Learning
  - Prediction of the unknown CRPs – 90% and up

- Side-Channel Analysis
  - Information leakage from the public helper data
  - Information leakage from power analysis
Oblivious RAM (ORAM)

- Encryption protect the data itself
- But data access patterns can still be learned
- Solution
  - Oblivious RAM
    - Any two access patterns of the same length are computational indistinguishable by anyone other than the client
    - Obfuscate the data access patterns
Oblivious RAM (ORAM)

- One approach
  - On each processor read or write bring the whole external memory to on-chip (i.e., client side)
  - More specifically
    - Encrypt all data, send to the untrusted environment, i.e., server side
    - On read or write bring all back, decrypt all, then pick the one that you want
    - Note that you can just pick and decrypt the one that you need and keep the rest unchanged
Oblivious RAM (ORAM)

- It is obvious that this is very expensive or even dreadfully inefficient
- So most of the research on ORAM is to find more efficient structures with comparable obfuscation capabilities
- The square-root algorithm
  - For each $\sqrt{N}$ accesses, permute the first $N + \sqrt{N}$ memory locations
  - $k$ steps of original RAM access can be simulated with $k + \sqrt{N}$ steps in the ORAM
- Hierarchical ORAM
  - Use a hierarchy of buffers, i.e., hash tables of different sizes scheme
  - General ideal
    - Server
      - $\log N$ levels for $N$ items, where level $i$ contains $2^i$ buckets and each bucket contains $\log N$ slots
    - Client
      - Pseudo Random Permutation (PRP) key $i$ for each level
Oblivious RAM (ORAM)

How does it work?

- Data are organized in blocks and each block is paired with a unique ID forming an item
  - Item = {block, id}
- System capacity
  - The total number of items in the system
- Server
  - Used to perform the general key-value storage service
- Functions
  - get(k) to get a value to a specific key
  - put(k, v) to put a value to a specific key
  - getRange(k₁, k₂, d) to return the first d items with keys in range [k₁, k₂]
  - delRange(k₁, k₂) : remove all items with keys within range [k₁, k₂]
- Client
  - Has a private memory
Oblivious RAM (ORAM)

- **Tree-based ORAM**
  - Organize data blocks on the server as a full binary tree
    - \( \log N \) levels and \( N \) leaf nodes
  - Each node in the tree is a bucket of \( Z \) items
  - Each item is assigned to a random leaf node of the tree
  - There is a position map to track which leaf node is assigned to a data item
Oblivious RAM (ORAM)

- Tree-based ORAM
- Item $i$ is stored in the path starting from the tree root to leaf node position map $[i]$.
- Get the whole path that may contain the item.
- Put all items on the path in the cache on the client side.
### Oblivious RAM (ORAM)

<table>
<thead>
<tr>
<th>ORAM Construction</th>
<th>Computation Overhead</th>
<th>Cloud Storage</th>
<th>Communication Round</th>
<th>Client Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Amortized</td>
<td>Worst-Case</td>
<td>Amortized</td>
<td>Worst-Case</td>
</tr>
<tr>
<td>Basic-SR</td>
<td>(O(n \log n))</td>
<td>(O(\sqrt{N} \log N))</td>
<td>(O(N))</td>
<td>(O(\sqrt{N} \log N))</td>
</tr>
<tr>
<td>Basic-SR</td>
<td>(O(n \log^2 n))</td>
<td>(O(\sqrt{N} \log^2 N))</td>
<td>(O(N \log^2 N))</td>
<td>(O(\sqrt{N} \log^2 N))</td>
</tr>
<tr>
<td>IBS-SR</td>
<td>(O(\sqrt{N}))</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Basic-HR</td>
<td>(O(n \log n))</td>
<td>(O(\log^3 N))</td>
<td>(O(N \log^2 N))</td>
<td>(O(\log^3 N))</td>
</tr>
<tr>
<td>Basic-HR</td>
<td>(O(n \log^2 n))</td>
<td>(O(\log^4 N))</td>
<td>(O(N \log^3 N))</td>
<td>(O(\log^4 N))</td>
</tr>
<tr>
<td>BB-ORAM</td>
<td>Non-Recursive</td>
<td>(O(\log^2 N))</td>
<td>(O(N \log N))</td>
<td>(O(\log^2 N))</td>
</tr>
<tr>
<td>BB-ORAM</td>
<td>Recursive</td>
<td>(O(\log^3 N))</td>
<td>(O(N \log N))</td>
<td>(O(\log^3 N))</td>
</tr>
<tr>
<td>TP-ORAM</td>
<td>Non-Recursive, Non-Concurrent</td>
<td>(O(\log N))</td>
<td>(O(\sqrt{N}))</td>
<td>(O(\log N))</td>
</tr>
<tr>
<td>TP-ORAM</td>
<td>Non-Recursive, Concurrent</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>TP-ORAM</td>
<td>Recursive, Non-Concurrent</td>
<td>(O(\log^2 N \log B))</td>
<td>(O(\sqrt{N}))</td>
<td>(O(\log^2 N \log B))</td>
</tr>
<tr>
<td>TP-ORAM</td>
<td>Recursive, Concurrent</td>
<td>(O(\log^2 N \log B))</td>
<td>(O(\log^2 N \log B))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>TP-ORAM</td>
<td>Recursive, Non-Concurrent</td>
<td>(O(N \log N \log B))</td>
<td>(O(N \log N \log B + O(1)))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>TP-ORAM</td>
<td>Recursive, Concurrent</td>
<td>(O(N \log N \log B))</td>
<td>(O(N \log N \log B))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Path-ORAM</td>
<td>Non-Recursive</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Path-ORAM</td>
<td>Recursive</td>
<td>(O(\log^2 N \log B))</td>
<td>(O(\log^2 N \log B))</td>
<td>(O(N))</td>
</tr>
</tbody>
</table>
Circuit Obfuscation

- An algorithm or a circuit $\text{Obf}$ is an obfuscator if for any function, in software or hardware:
  - $\text{Obf}(f) = f$, i.e., $\text{Obf}(f)$ computes the same function as $f$
  - $\text{Obf}$ as an obfuscation operator is efficient if it runs in polynomial time, i.e., polynomial slowdown
Circuit Obfuscation

- **Hardness**
  - Obf(C) hard to analyze or reverse-engineer

- **Minimal requirement**
  - One-wayness
    - Hard to recover f from Obf(f)
  - Maximal requirement
    - Obf(f) is like a “black box” that evaluates f
Circuit Obfuscation

- K must be set properly for the circuit to function correctly
  - When K = 0, the XOR gate acts as buffer
  - When K = 1, then the XOR gate acts as inverter
- K can also be viewed as the key to the circuit
Circuit Obfuscation

- General-purpose obfuscation is impossible
- Hardware obfuscation aims to make it difficult for an adversary to understand or analyze the actual functionality of a design
- DSP Core Hardware Obfuscation
  - Structural Obfuscation
  - Functional Obfuscation
- Combinational/Sequential Hardware Obfuscation
Hardware Trojans

- Malicious changes to a design
- These changes be inserted at any stage of the design and manufacturing process
  - Specification stage, RTL, manufacturing, supply chain
- Often there are two components, a trigger and a payload
## Hardware Trojans

<table>
<thead>
<tr>
<th>Types of Trojan</th>
<th>Actor</th>
<th>Action</th>
<th>Input Channel</th>
<th>Output/Leaking channel</th>
<th>Payload / Consequence of attack</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trigger Activated</strong></td>
<td>Attacker with physical access to the device</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Particular legitimate input sequence</td>
<td>Standard Input</td>
<td></td>
<td>Leaking sensitive information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Particular illegitimate input sequence</td>
<td></td>
<td>Standard / Unused Outputs</td>
<td>Encryption Key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Taking control through unused functional units or interfaces</td>
<td>Unused Inputs</td>
<td></td>
<td>Plain text</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Standard Input</td>
<td></td>
<td>Denial of service</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Generating incorrect results</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Make the device stop working</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reduce the reliability of the device</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Side Channels</td>
<td>Drain the battery</td>
</tr>
<tr>
<td><strong>Legitimate User</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Normal operation for certain n&gt;N</td>
<td>Standard Input</td>
<td></td>
<td>Leak the Encryption Key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Particular legitimate input sequence</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Illegitimate input sequence by mistake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Certain time interval between two legal inputs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Always Active</strong></td>
<td>N/A</td>
<td>N/A</td>
<td>Internal IP Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Side Channels</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EM Waves</td>
</tr>
</tbody>
</table>

---

**References:**

1. Y. Jin, “Experiences in Hardware Trojans Design and Implementation”
2. G. Becker, “Implementing Hardware Trojans”