# Stealthy Opaque Predicates in Hardware -
## Obfuscating Constant Expressions at Negligible Overhead


Max Hoffmann, Christof Paar

Ruhr University Bochum, Horst-Görtz Institute for IT-Security, Germany



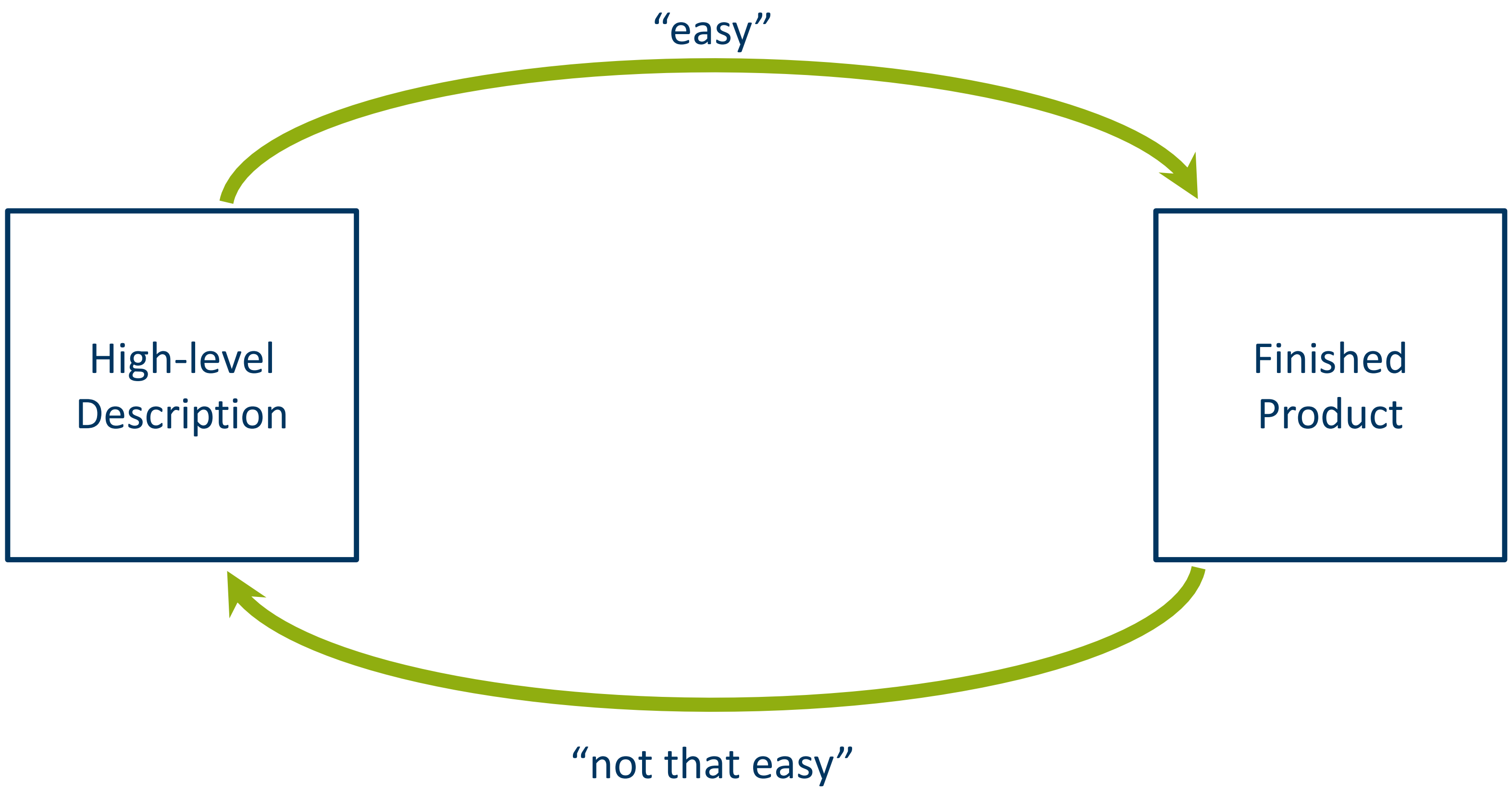
CHES 2018 | Amsterdam 10.09.2018


Horst Görtz Institut für IT-Sicherheit

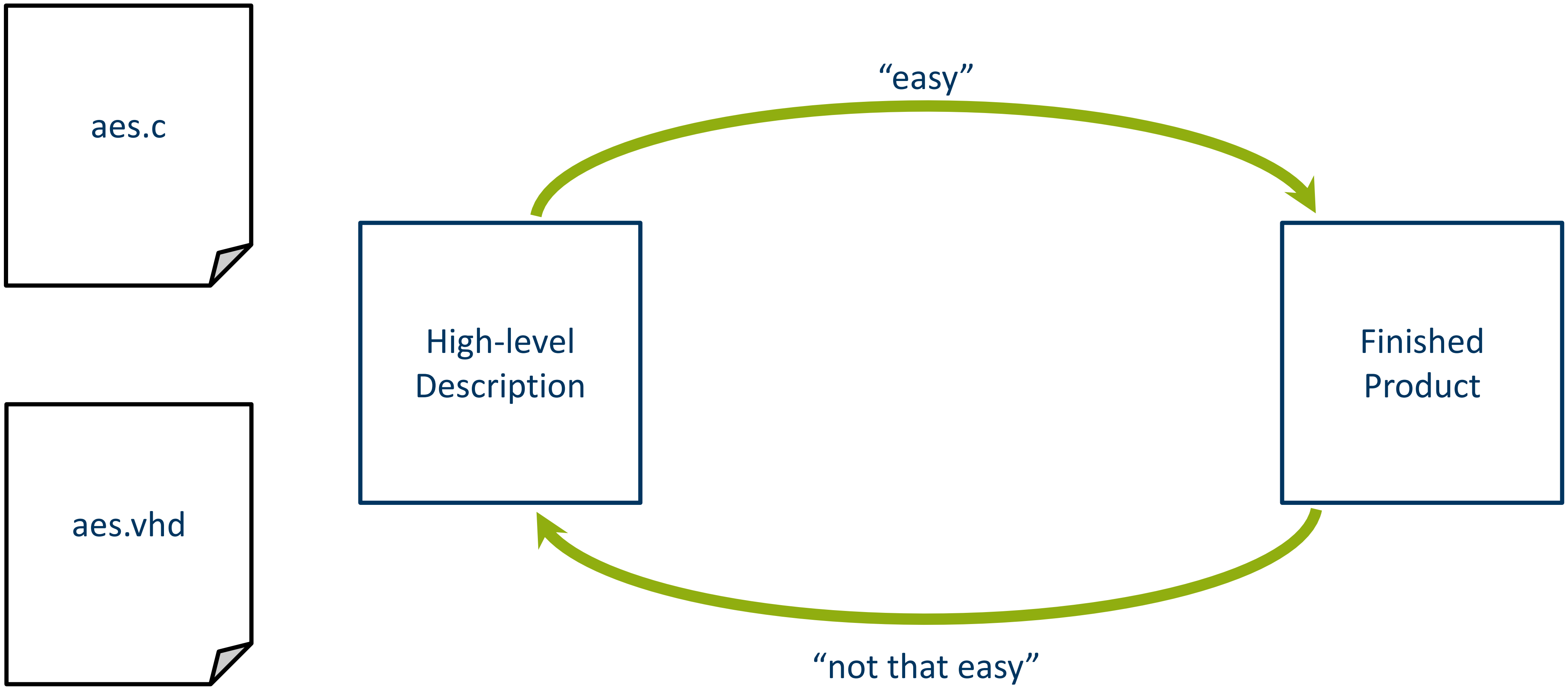
# Obfuscation

Source: www.constructionknowledge.net

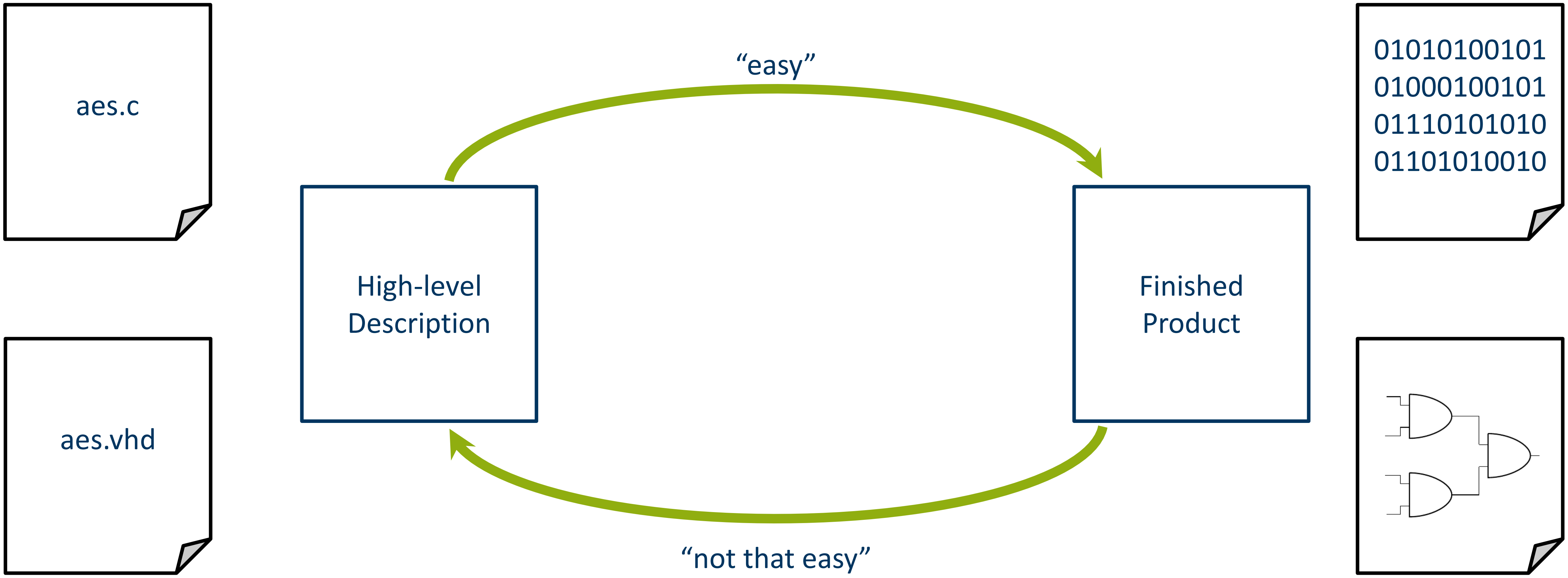
# Why Obfuscation?



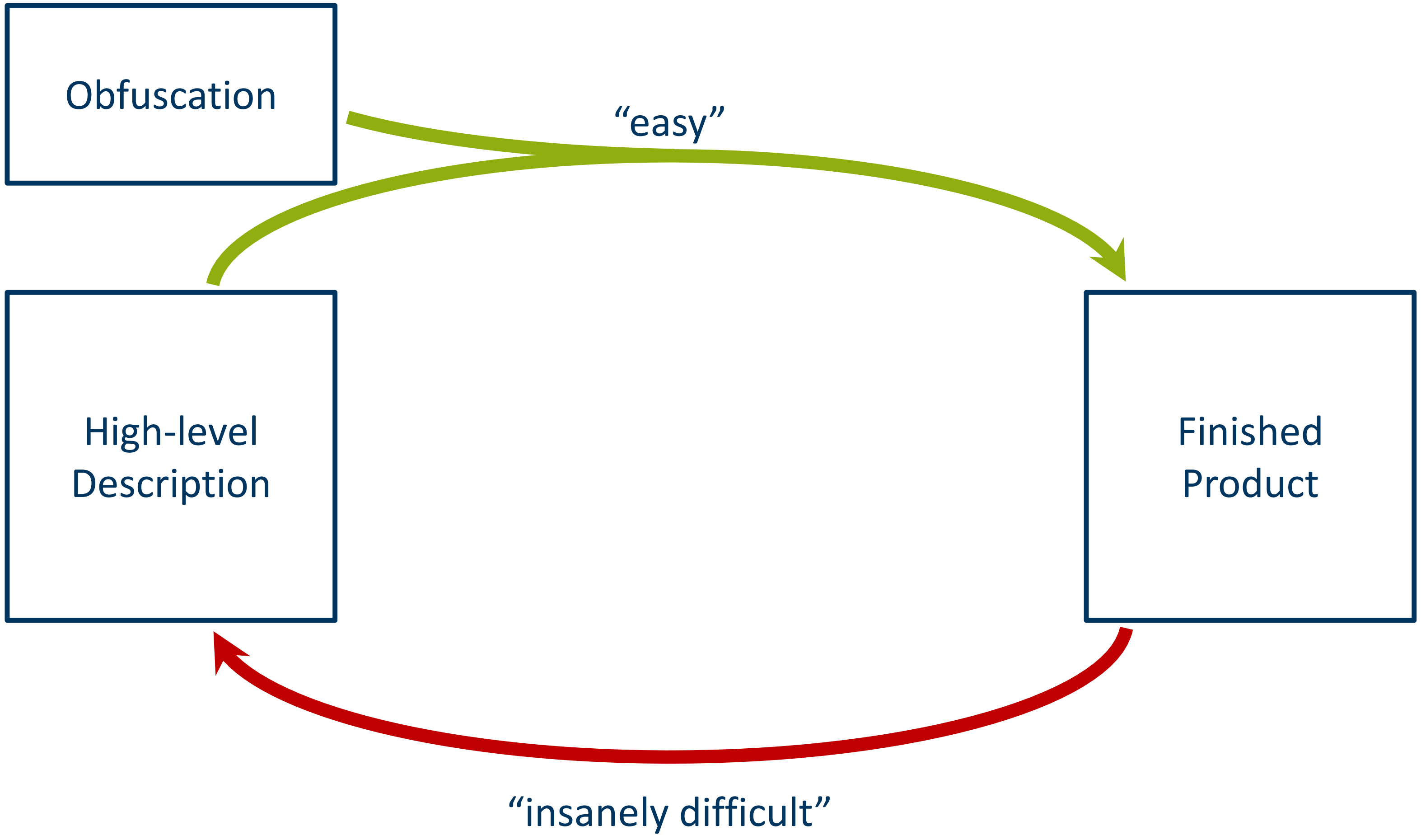
# Why Obfuscation?



# Why Obfuscation?



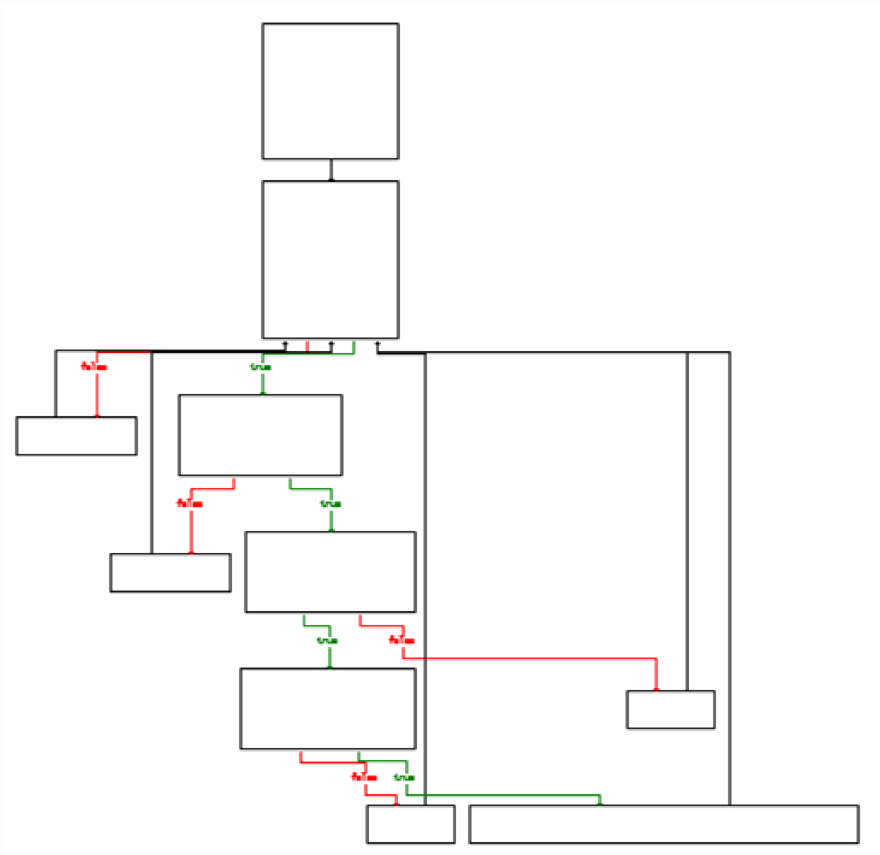
# Why Obfuscation?



# Software Obfuscation

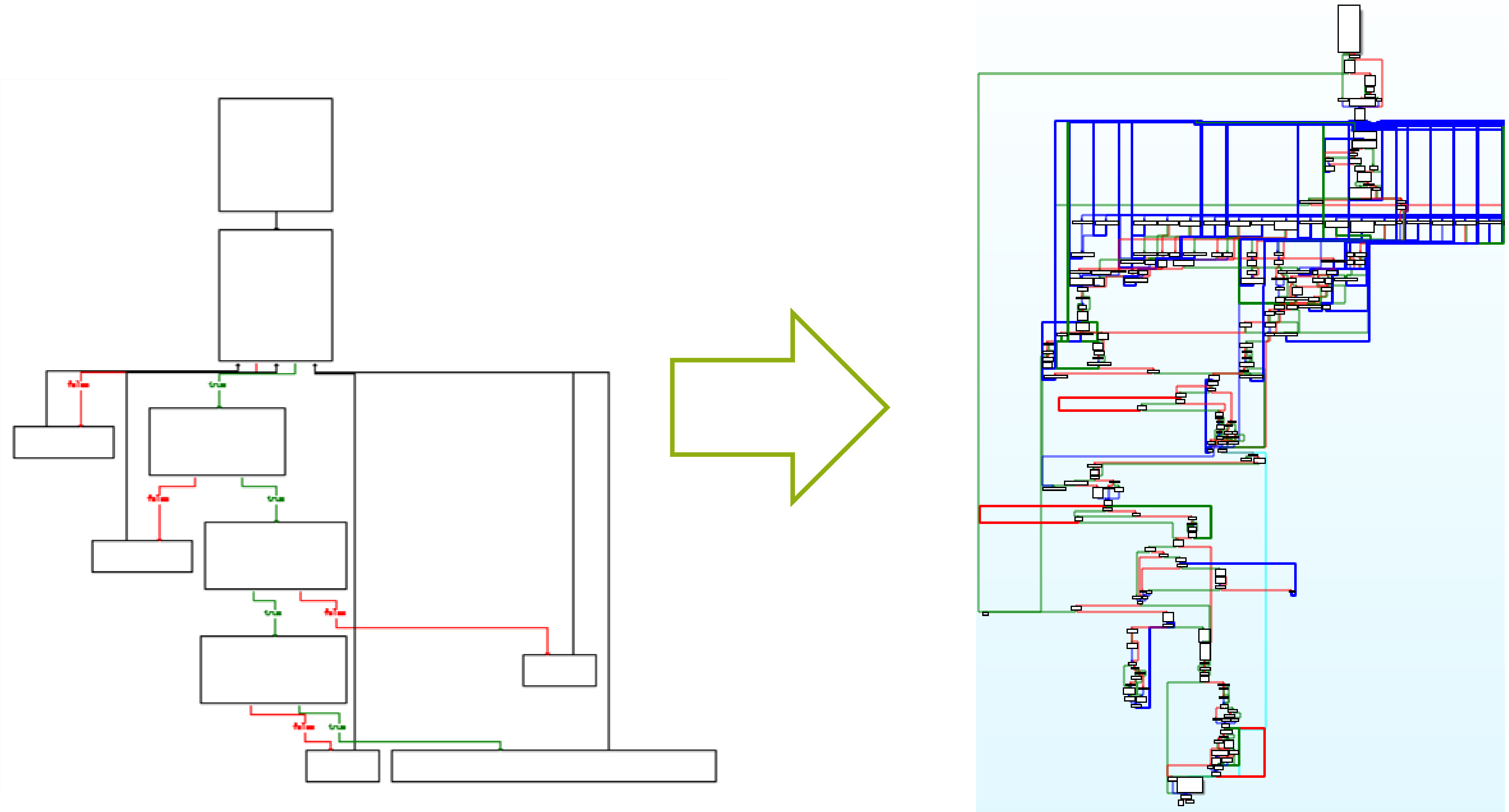


- One target in software is control flow obfuscation.



# Software Obfuscation

- One target in software is control flow obfuscation.



# Software Obfuscation

- Opaque Predicates are used as a basic building block.

# Software Obfuscation

- Opaque Predicates are used as a basic building block.

- An opaque predicate:
  - is an expression
  - looks like having a dynamic value
  - evaluates to a constant, known value

Example:
```
(x * (x + 1)) % 2 == 0
```

# Software Obfuscation

- Opaque Predicates are used as a basic building block.

- An opaque predicate:
  - is an expression
  - looks like having a dynamic value
  - evaluates to a constant, known value

Example:
```
(x * (x + 1)) % 2 == 0
```

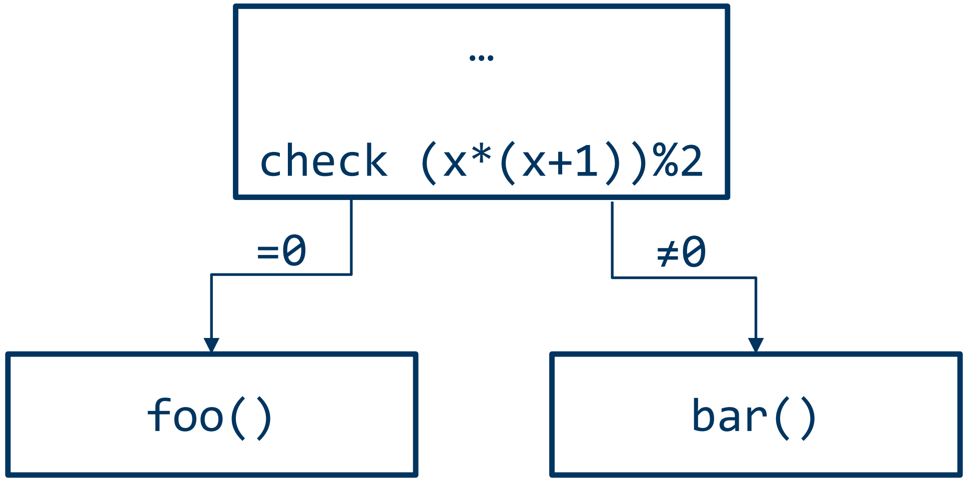
- Meant to harden against static analysis.

> - **Static Analysis**: analysis performed solely on a static data, e.g., a binary.
> - **Dynamic Analysis**: analysis performed during operation, e.g., while executing a binary.

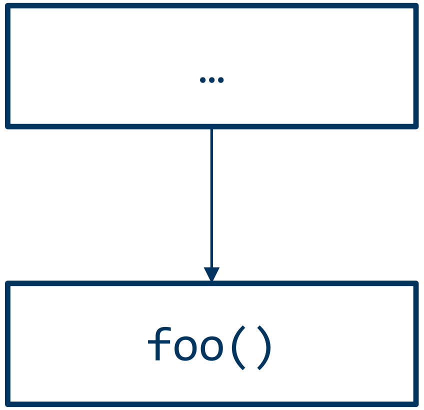
# Example: Software Opaque Predicates

```
if ((x * (x + 1)) % 2 == 0):
   foo()
else
   bar()
```

- Control flow graph of a static analyzer:




- "True" control flow graph:



# A Software Obfuscation Technique in Hardware?

RUB

- How can a software obfuscation technique help in hardware?
- Obfuscation should harden against reverse engineering.

# A Software Obfuscation Technique in Hardware?

RUB

- How can a software obfuscation technique help in hardware?
- Obfuscation should harden against reverse engineering.
- Reverse engineers rarely analyze an entire design.
- Mostly: small parts of a design.

# A Software Obfuscation Technique in Hardware?

RUB

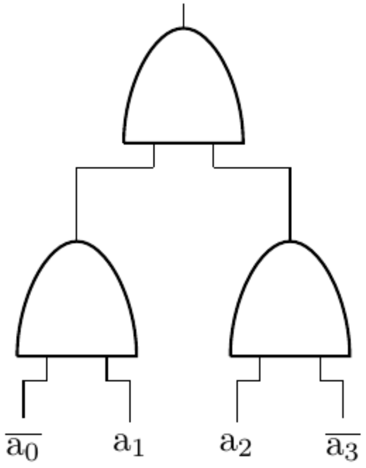
- How can a software obfuscation technique help in hardware?
- Obfuscation should harden against reverse engineering.
- Reverse engineers rarely analyze an entire design.
- Mostly: small parts of a design.
- **Goal**: hide as much information as possible.
  → reduces starting points for reverse engineers.
  → makes understanding of any component harder.

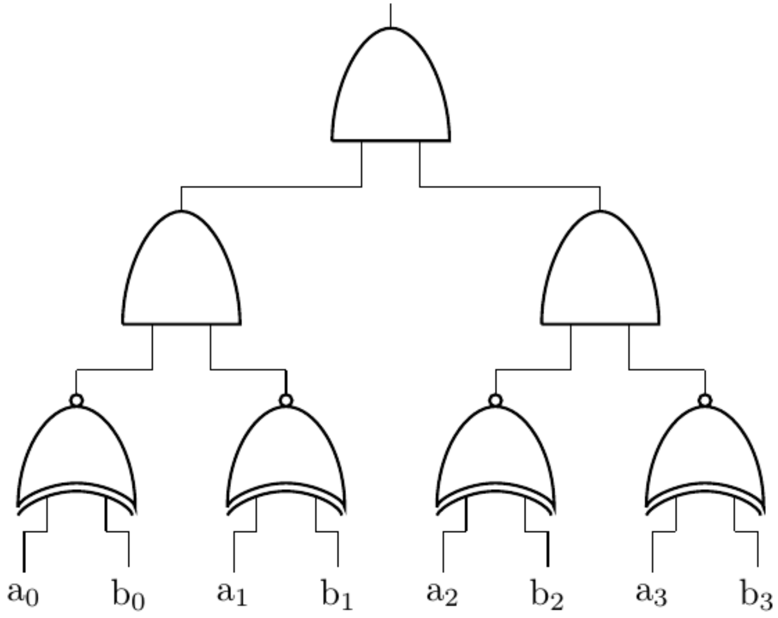
# Example: Hardware Reversing

```
if a = "0110" then
    output <= ‘1’;
end if;
```

vs.

```
if a = b then
    output <= ‘1’;
end if;
```



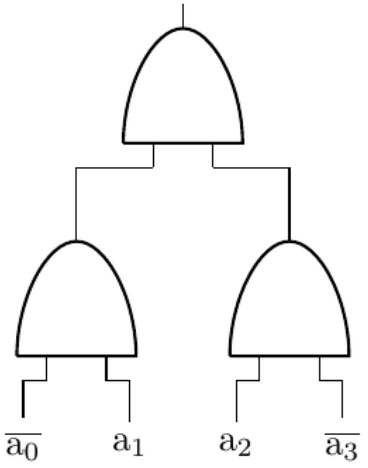



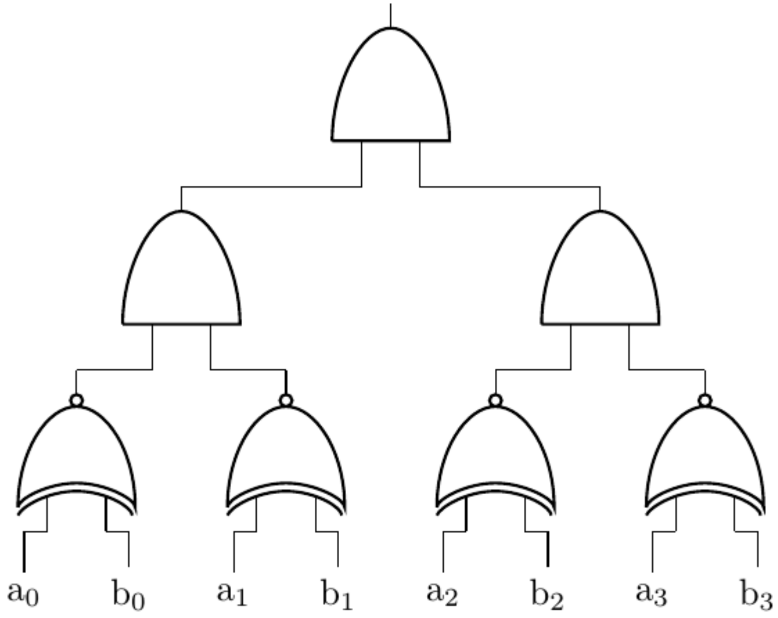
# Example: Hardware Reversing

```
if a = "0110" then
    output <= ‘1’;
end if;
```

vs.

```
if a = b then
    output <= ‘1’;
end if;
```



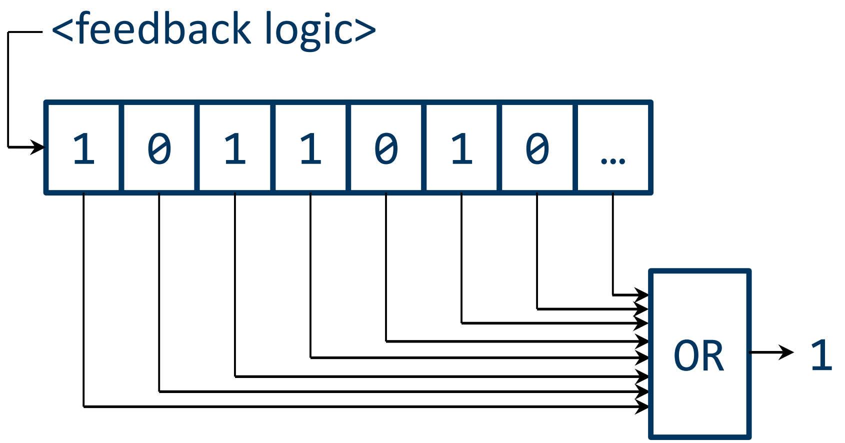




→ Use OPs to hide information introduced by constant signals.

# PREVIOUS WORK

# Translation to Hardware

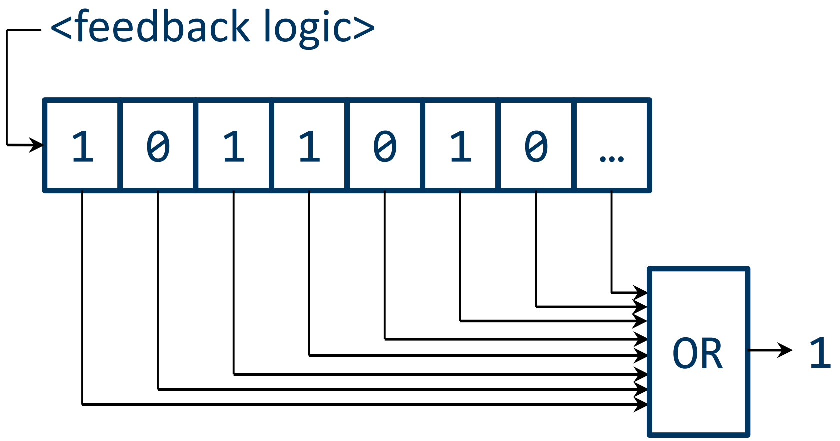
- Only one prior work on opaque predicates.
- Sergeichik et al. presented LFSR-based OPs in 2014 [1].




[1] Sergeichik and Ivaniuk. "Implementation of opaque predicates for fpga designs hardware obfuscation." (JICMS, 2014).

# Stealthiness

- **Problem**: Easy to detect, uncommon structure
- Removal via static analysis demonstrated in [1].



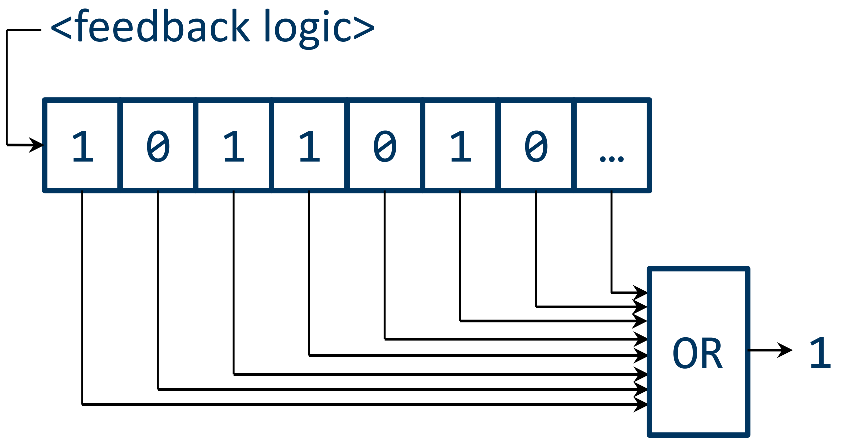

[1] Wallat, Fyrbiak, Schlögel, and Paar. “A Look at the Dark Side of Hardware Reverse Engineering – A Case Study” (*IVSW*, 2017)

# Stealthiness

RUB

- **Problem**: Easy to detect, uncommon structure

- Removal via static analysis demonstrated in [1].




- **Desired Metric**: “Stealthiness“
  - Impossible (?) to measure
  - Human factor plays a role
  - Different in hardware and software

[1] Wallat, Fyrbiak, Schlögel, and Paar. “A Look at the Dark Side of Hardware Reverse Engineering – A Case Study” (*IVSW*, 2017)

# Opaque Predicates in Hardware

# Hardware OPs – Idea

- Stealthiness: use common structures.

- Try to use existing circuitry.

# Hardware OPs – Idea

- Stealthiness: use common structures.

- Try to use existing circuitry.

- **Observation**:
  - Signals are changing constantly.
  - A signal's value is only important while evaluated.

# Hardware OPs – Idea

RUB

- Stealthiness: use common structures.

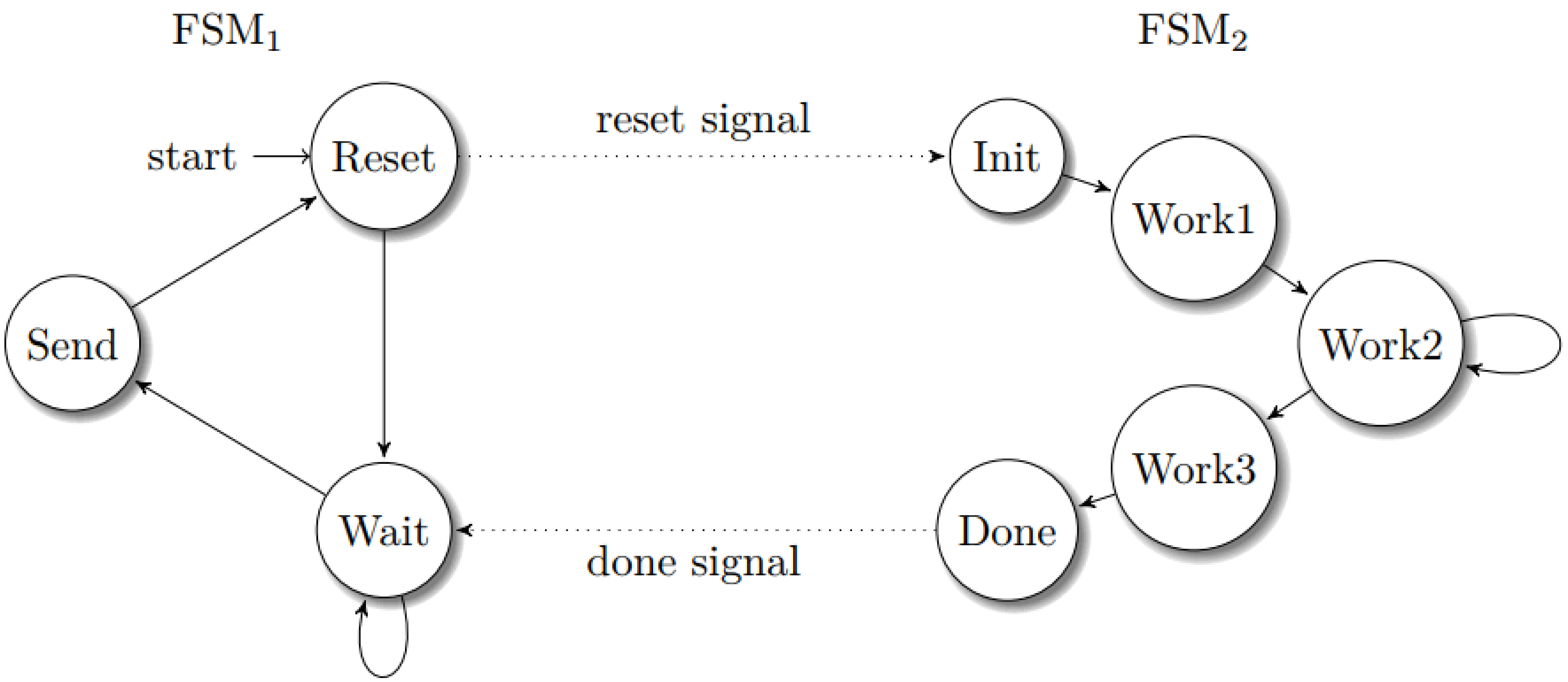
- Try to use existing circuitry.

- **Observation**:
  - Signals are changing constantly.
  - A signal's value is only important while evaluated.

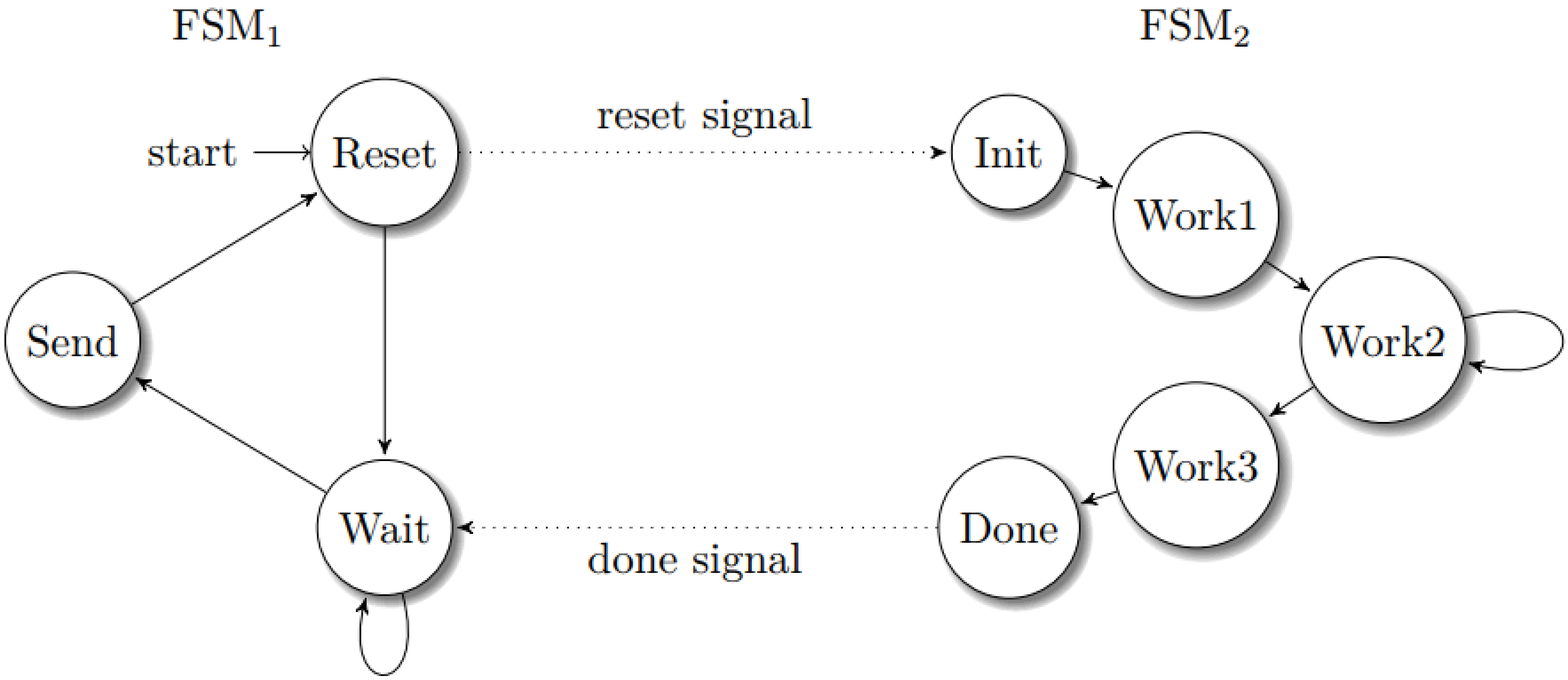
→ Use an existing signal which

1. has the required state whenever we need it
2. switches "randomly" when not needed.

# Example: Hardware OPs

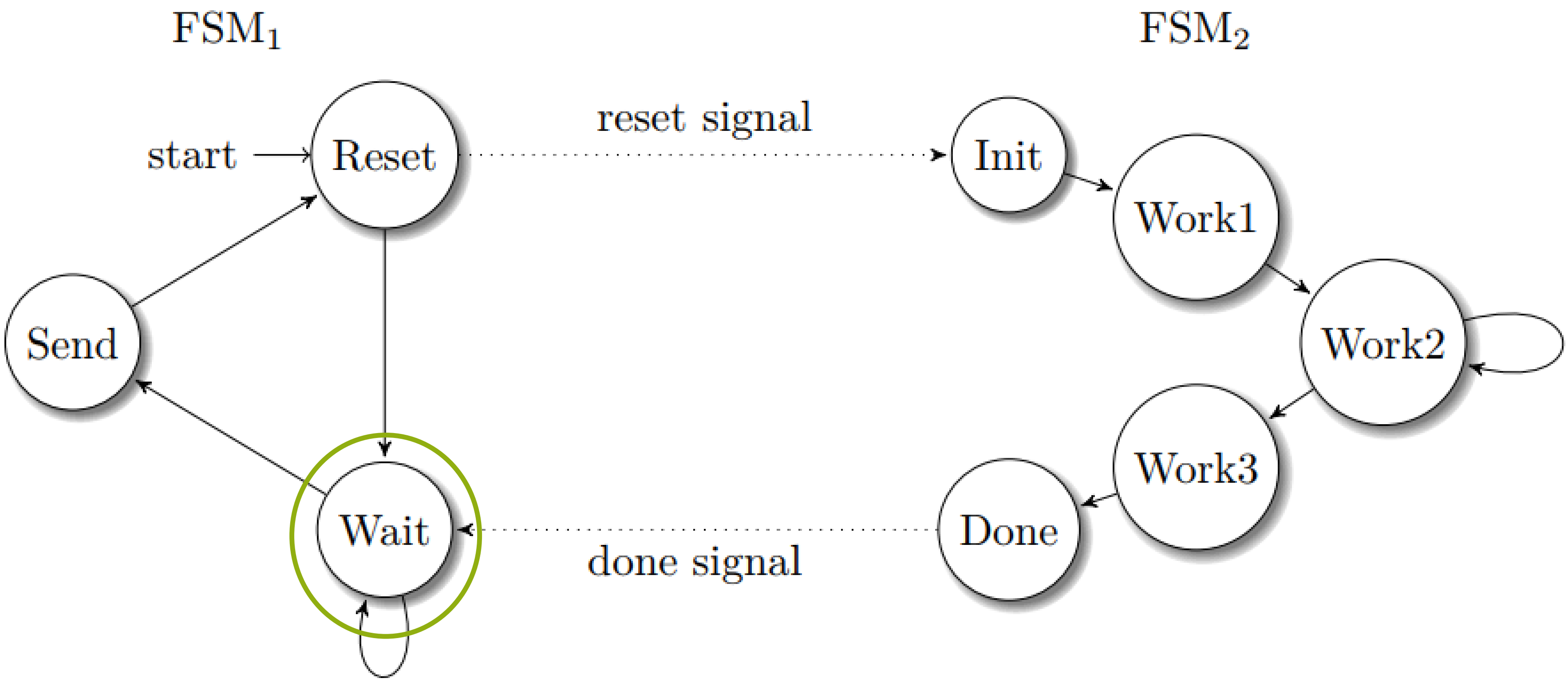


# Example: Hardware OPs



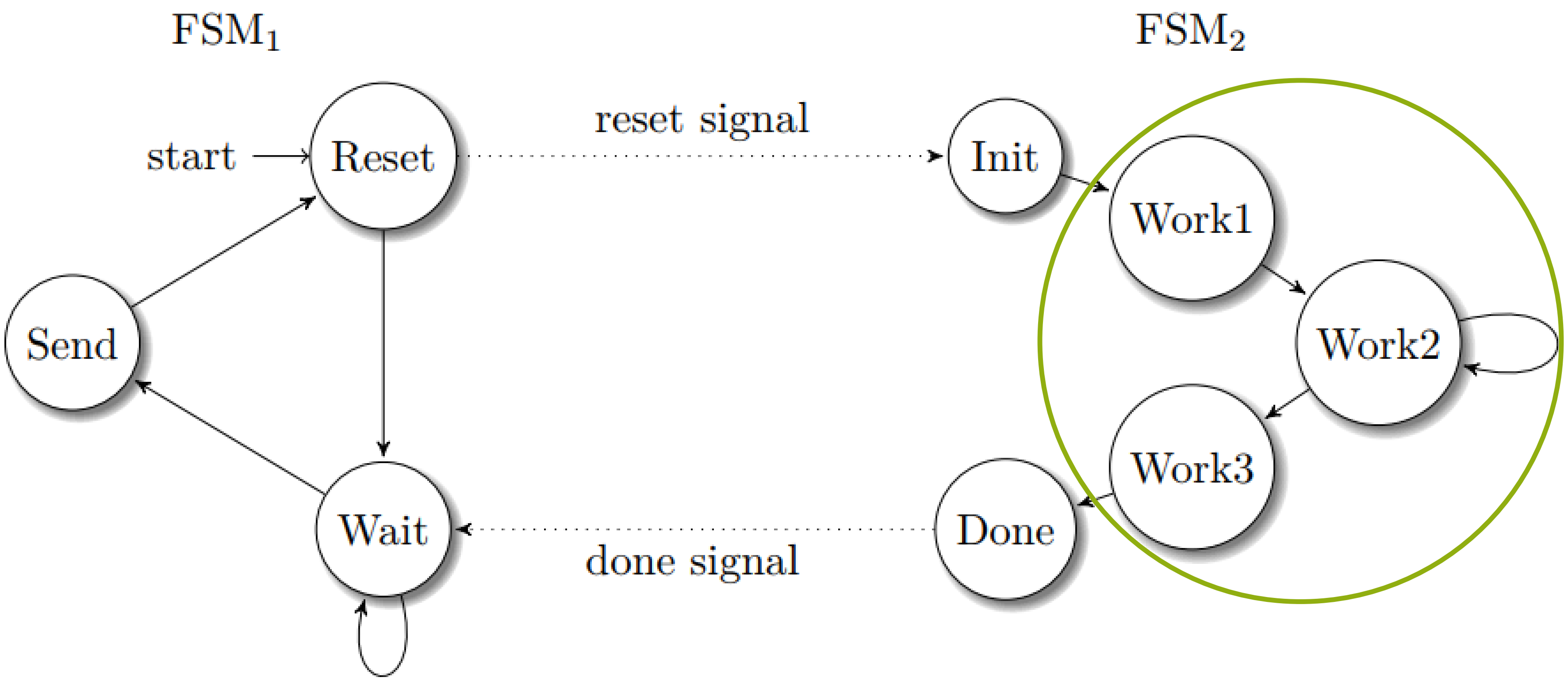

- Constant value required in Work1, Work2, and Work3.
- Multiple options to use the state of an FSM as an OP.

# Example: Hardware OPs




- Constant value required in Work1, Work2, and Work3.
- Multiple options to use the state of an FSM as an OP.

# Example: Hardware OPs

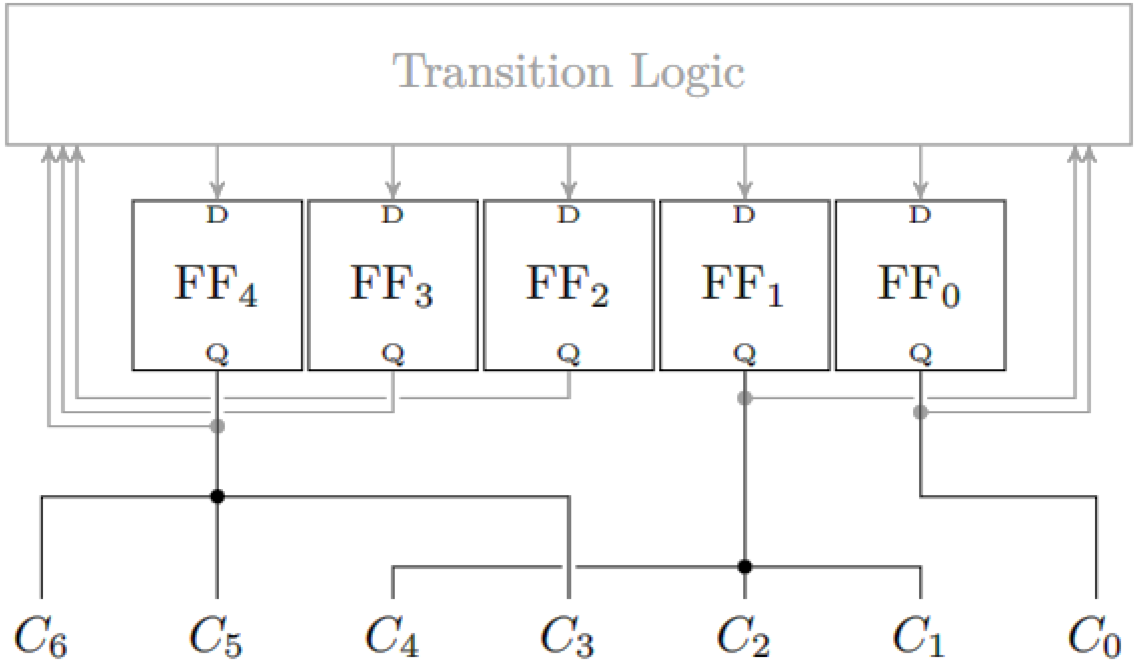



- Constant value required in Work1, Work2, and Work3.
- Multiple options to use the state of an FSM as an OP.

# Hardware OPs

RUB

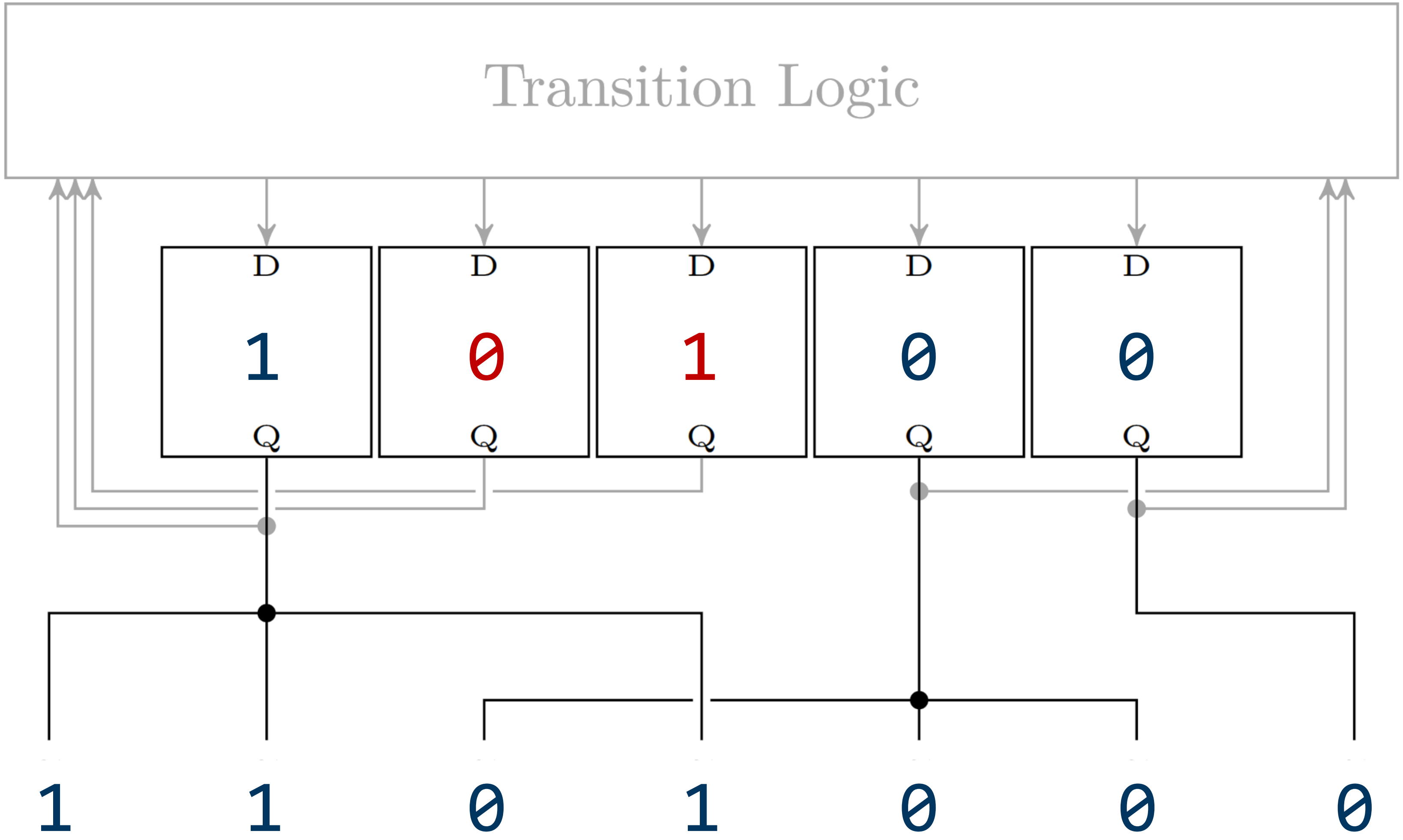
- Example:
  - Constant $1101000_2$ to be obfuscated.
  - 5-bit FSM passes 3 states during the processing period.



# Hardware OPs

RUB

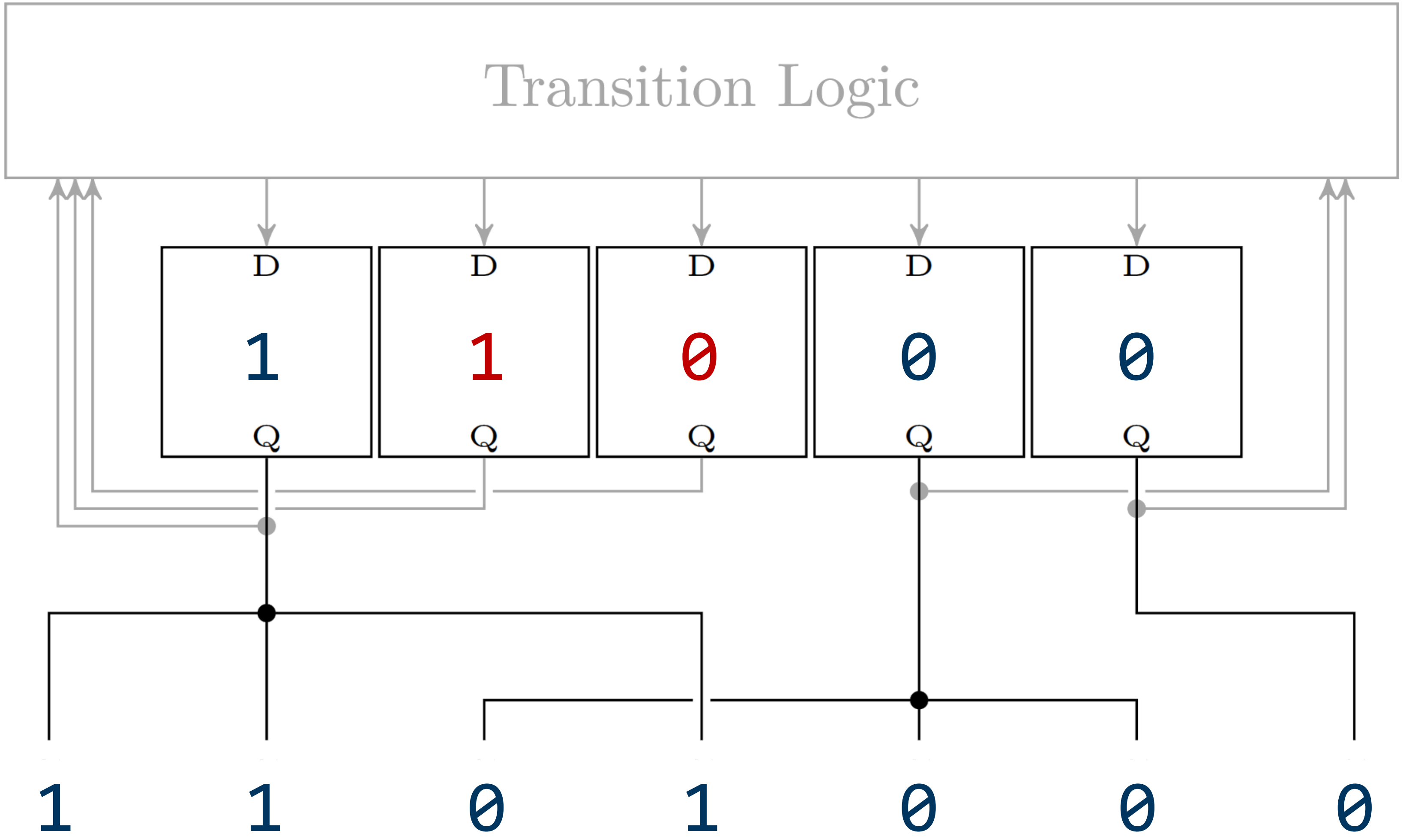
- $1^{st}$ State:



# Hardware OPs

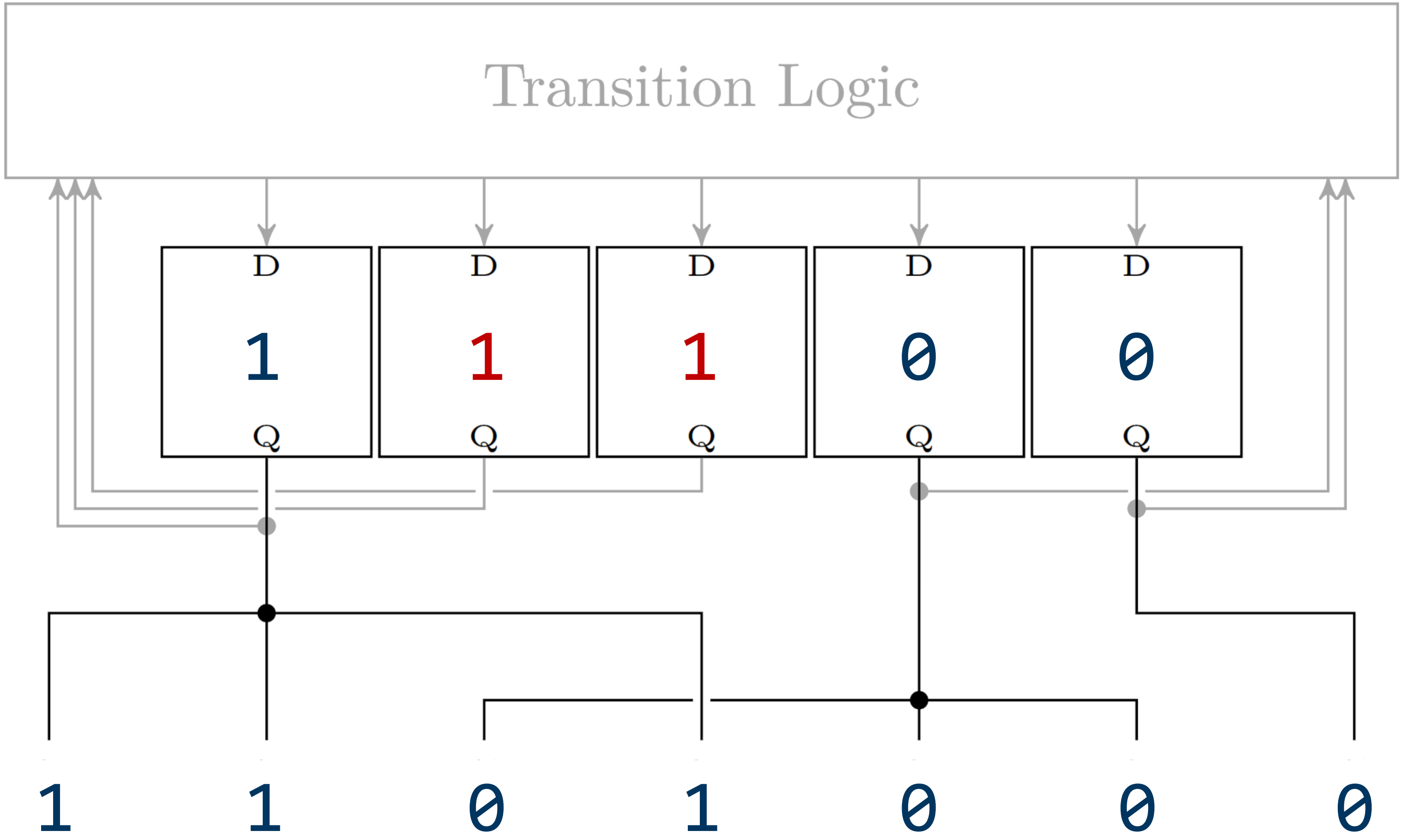
RUB

- 2nd State:



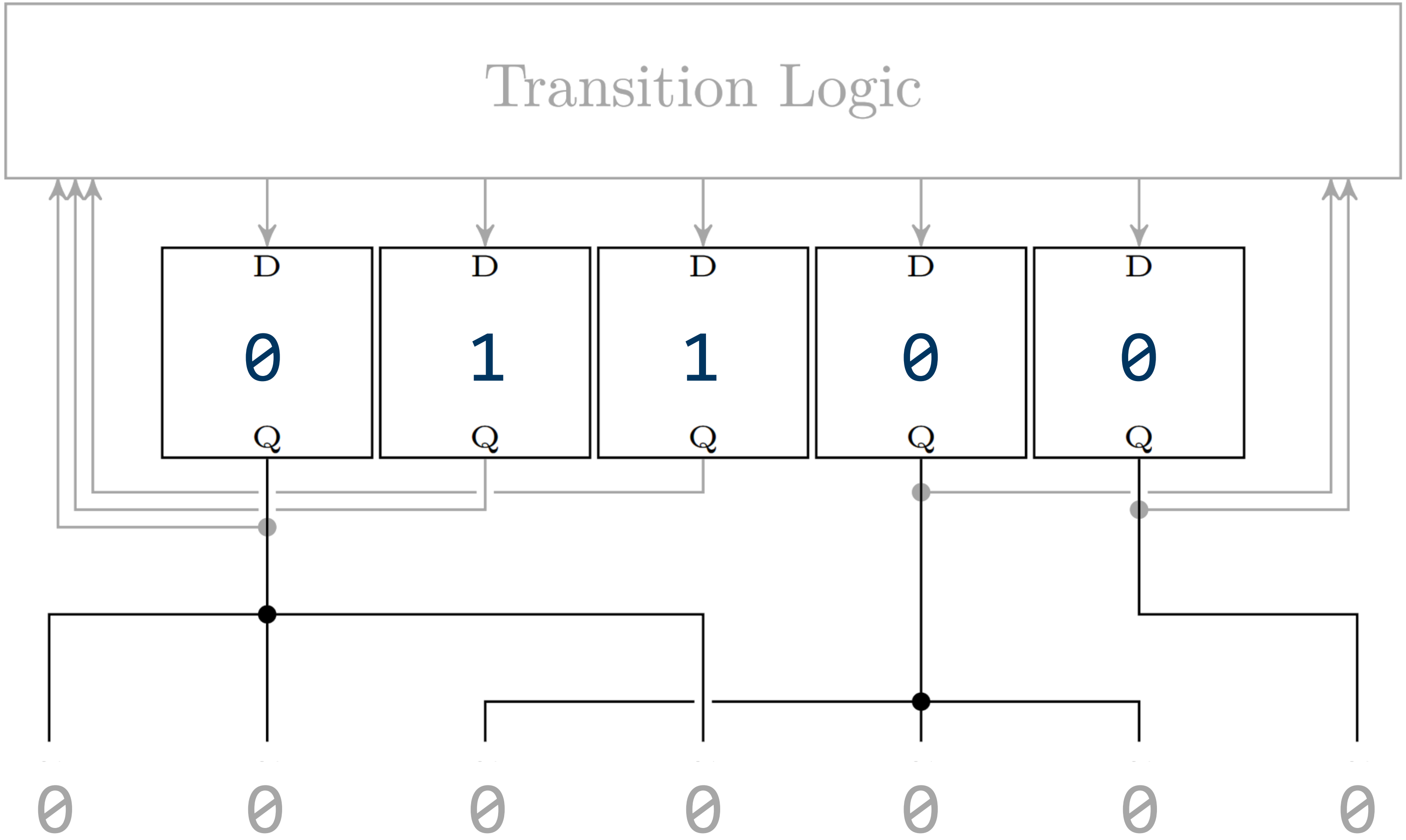
# Hardware OPs

- 3rd State:



# Hardware OPs

- 4th State:



# Hardware OPs

- Very stealthy: existing FSMs are used.
- Zero additional gates (in theory…)

# Hardware OPs

- Very stealthy: existing FSMs are used.
- Zero additional gates (in theory…)
- Applicable to nearly all designs.
- Considerably increases reversing effort:
  Reversing of control- and data-path required for identification of constants.

# Hardware OPs

RUB

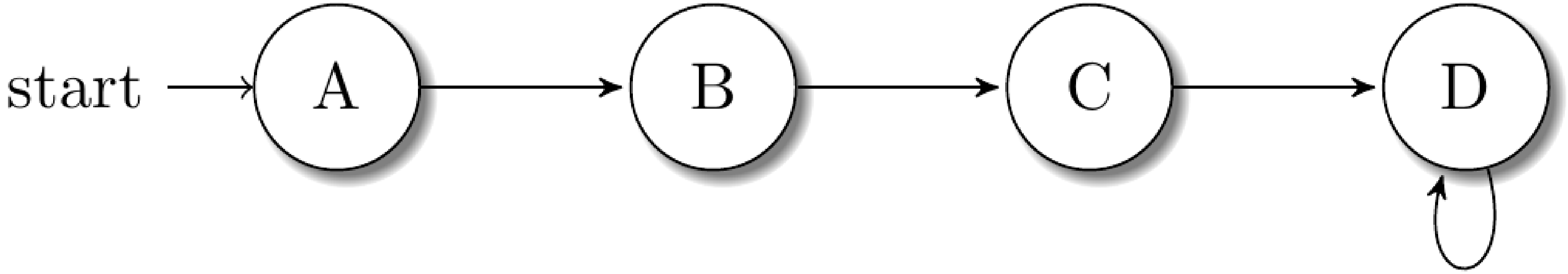
- Very stealthy: existing FSMs are used.
- Zero additional gates (in theory...)
- Applicable to nearly all designs.
- Considerably increases reversing effort:
  Reversing of control- and data-path required for identification of constants.
- Applicable to ASICs and FPGAs.
- Forces a reverse engineer to apply dynamic analysis.

# Hardware OPs

- If no suitable FSM available, add a new FSM-like module.
  - Make it reset outside of the processing period.
  - Make it stabilize in a known state after some cycles.
  - Generate OP value from stable state.

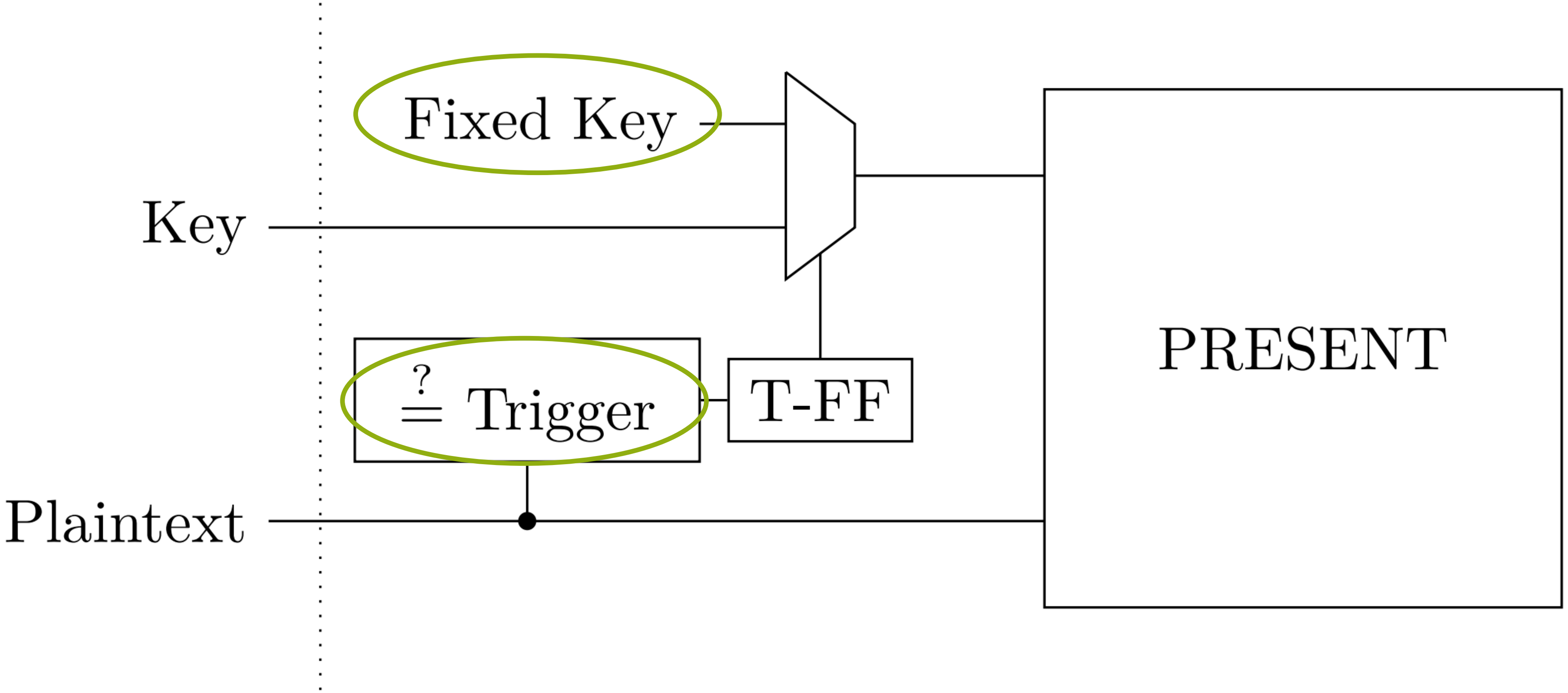
- Still stealthy (FSMs are common).

- Stabilizing FSMs are also common (DONE state).



# Case Studies

# Scenario



# Scenario

RUB

**Algorithm 1** Subverted RSA KeyGen

**Input:** $1^\lambda$

**Output:** $\text{pk} = (n, e)$, $\text{sk} = (d)$

1: Choose $p, q$ as random $\lambda/2$-bit primes

2: $n \leftarrow pq$

3: $e \leftarrow p^{E_{adv}} \mod N_{adv}$

4: **while** $\gcd(e, \Phi(n)) \neq 1$ **do**

5: $\quad e \leftarrow e + 1$

6: $d \leftarrow e^{-1} \mod \Phi(n)$

7: **return** $\text{pk} = (n, e)$, $\text{sk} = (d)$

# Results

RUB

| Design | | LUTs | | FFs | |
|---|---|---|---|---|---|
| PRESENT | Unobfuscated | 304 | | 347 | |
| | Strategy 1 | 307 | +0.99% | 347 | +0% |
| | Strategy 2 | 304 | +0% | 350 | +0.86% |
| RSA | Unobfuscated | 10570 | | 5316 | |
| | Strategy 1 | 10811 | +2.28% | 5314 | −0.04% |
| | Strategy 2 | 10692 | +1.15% | 5323 | +0.13% |

**Platform**: XILINX Artix-7 35T FPGA

**Legend**:

Unobfuscated: no opaque predicates were used

Strategy 1: opaque predicate from existing circuitry

Strategy 2: new circuitry for the opaque predicate

# APPLICATION: WATERMARKING

# Watermarking

RUB

- A watermark enables identification of IP-theft.
- A vendor can inspect products for presence of his watermark.
- Schmid et al. proposed a watermarking scheme for FPGAs which implements a watermark into LUT configurations [1].

## Stealthy Opaque Predicates in Hardware - Obfuscating Constant Expressions at Negligible Overhead


Max Hoffmann and Christof Paar

Horst Görtz Institute for IT-Security, Ruhr-Universität Bochum, Germany,
{max.hoffmann,christof.paar}@rub.de



**Abstract.** Opaque predicates are a well-established fundamental building block for software obfuscation. Simplified, an opaque predicate implements an expression that provides constant Boolean output, but appears to have dynamic behavior for static analysis. Even though there has been extensive research regarding opaque predicates in software, techniques for opaque predicates in hardware are barely explored. In this work, we propose a novel technique to instantiate opaque predicates in hardware, such that they (1) are resource-efficient, and (2) are challenging to reverse engineer even with dynamic analysis capabilities. We demonstrate the applicability of opaque predicates in hardware for both, protection of intellectual property and obfuscation of cryptographic hardware Trojans. Our results show that we are able to implement stealthy opaque predicates in hardware with minimal overhead in area and no impact on latency.



**Keywords:** Opaque Predicates · Obfuscation · Hardware Reverse Engineering · Hardware Analysis


### 1 Introduction

The process of designing, implementing and testing hardware cores is both time consuming and costly. Thus, in today's hardware design process, not every piece of a product is created anew. Many products consist partially of Intellectual Property (IP)-cores, i.e. third party modules. The market for IP-cores is large with USD 2.45 billion in 2013 [1] and was recently "expected to be worth USD 6.22 billion by 2023" [2]. IP-cores are sold as either *soft-cores* in the form of HDL code, *firm-cores* as a netlist, or *hard-cores* as a fixed layout for a certain technology. Preferably they are sold as hard- or firm-cores with only the interfaces and functional specifications known to maintain exclusivity and hinder IP-theft. Consequently, the IP-core market suffers from copyright infringements as code can illegally be copied without relicensing. Additionally, IP-cores have become a major target for reverse engineering in order to understand technologies of competitors and to find hidden backdoors.

Protecting hardware designs has therefore moved into the focus of the design industry. To counter IP-theft, vendors include watermarks into their products. A watermark is a vendor- or customer-specific attribute, embedded into the design. It does not impact functionality but its existence can easily be verified by the vendor. Consequently, the vendor is able to analyze products on the market for license violations. However, since such a watermark is often a static part of the design (c.f. [3, 4]) or mostly disconnected from the rest of the circuitry (c.f. [5]), it can be found by leveraging these properties and subsequently removed by reverse engineers. To counter such removal attacks and harden against reverse engineering in general, designers can use obfuscation methods.




IACR Transactions on Cryptographic Hardware and Embedded Systems Vol. 0, No.0, pp.1—21, DOI:XXXXXXXX




[1] Schmid, Moritz, and Ziener, Daniel, and Teich, Jurgen. "Netlist-level IP protection by watermarking for LUT-based FPGAs." (FPT 2008)

# FPGA LUT Configuration

RUB

- A LUT is configured by defining it's output values.

- Example:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $I_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $I_2$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $I_1$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $I_0$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| output/config | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |

- These configurations can be read from the bitstream of an FPGA.

# Watermarking by Schmid et al.

RUB

- **Idea**: fix some inputs to GND.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\text{GND} \rightarrow I_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $\text{GND} \rightarrow I_2$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $I_1$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $I_0$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

# Watermarking by Schmid et al.

- **Idea**: fix some inputs to GND.

- Configuration bits for other cases become effectively unused.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathrm{GND} \rightarrow I_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $\mathrm{GND} \rightarrow I_2$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $I_1$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $I_0$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

# Watermarking by Schmid et al.

RUB

- **Idea**: fix some inputs to GND.
- Configuration bits for other cases become effectively unused.
- Embed watermark there.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GND $\rightarrow I_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| GND $\rightarrow I_2$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $I_1$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $I_0$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| output/config | C | C | C | C | W | W | W | W | W | W | W | W | W | W | W | W |

# Applying OPs

- Netlist-level attacker was included in attacker model.

- **Problem**: Tracing GND to LUTs → detected → easy to remove the watermark.

# Applying OPs

- Netlist-level attacker was included in attacker model.
- **Problem**: Tracing GND to LUTs → detected → easy to remove the watermark.
- **Solution**: Use our OPs instead of GND.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP $\rightarrow I_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| OP $\rightarrow I_2$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $I_1$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $I_0$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| output/config | C | C | C | C | W | W | W | W | W | W | W | W | W | W | W | W |

# Conclusion

# Conclusion

RUB

- Novel technique for opaque predicates in hardware (ASICs + FPGAs).
- Strong technique (discussion in the paper).
- Instantiation strategies:
  - Existing circuitry.
  - Additional circuitry.
- Practical evaluation.
- Demonstrate potential to mitigate existing attacks.

# Thank You For Your Attention!
# Any Questions?



Horst Görtz Institut
für IT-Sicherheit