Intro. to Computer Architecture

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

January 9, 2018

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

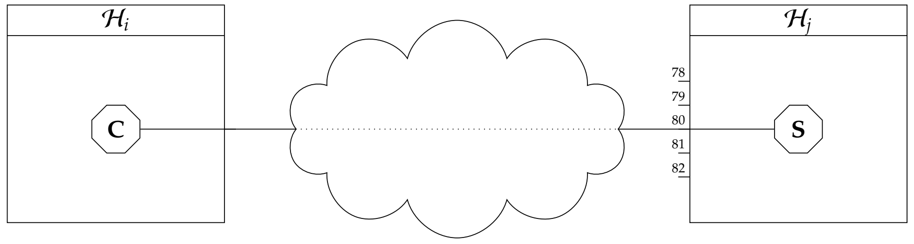
COMS12200 lecture: week #3

- Agenda:
  1. comments, questions, recap, then
  2. some use-cases for Boolean algebra as applied in **cryptography**.

Notes:

Boolean algebra ⇝ the PRESENT block cipher (2)

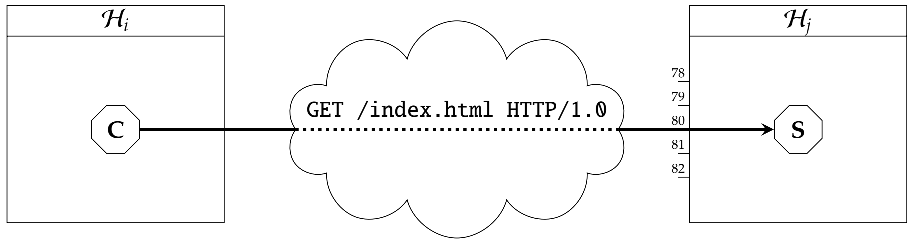
- Problem: confidential (bulk) communication, per



Notes:

Boolean algebra ⇝ the PRESENT block cipher (2)

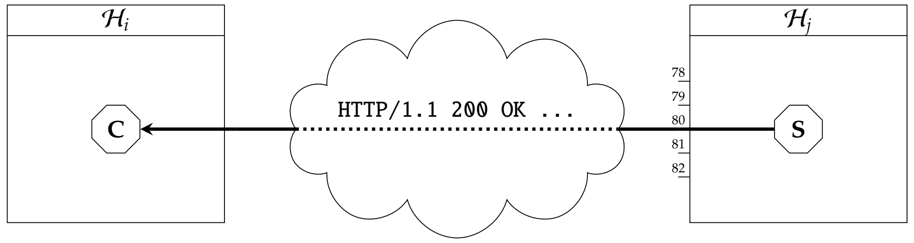
- Problem: confidential (bulk) communication, per



Notes:

Boolean algebra ⇝ the PRESENT block cipher (2)

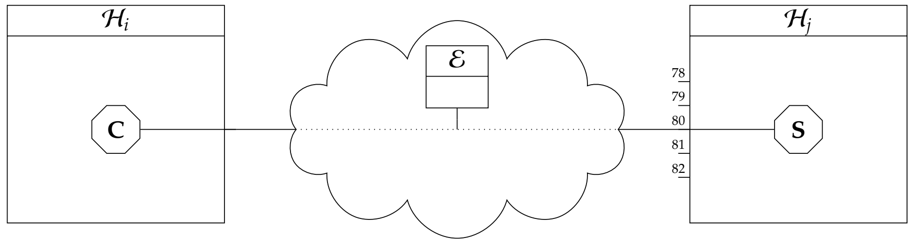
- Problem: confidential (bulk) communication, per



Notes:

Boolean algebra ⇝ the PRESENT block cipher (2)

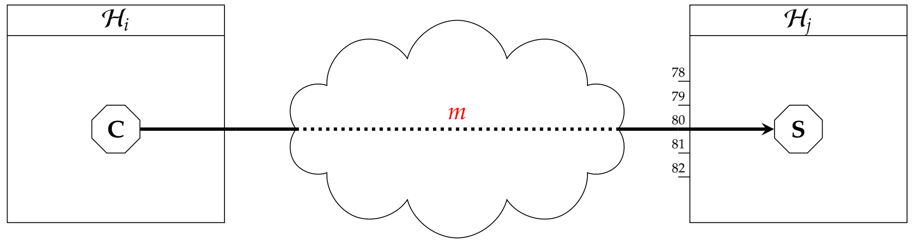
- Problem: confidential (bulk) communication, per



Notes:

Boolean algebra ⇝ the PRESENT block cipher (2)

- Problem: confidential (bulk) communication, per

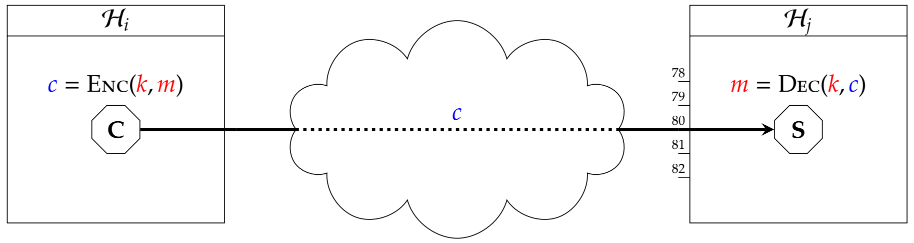


- Solution: **encryption**.
  - one option is to use a **block cipher**, which assumes $\mathcal{H}_i$ and $\mathcal{H}_j$ know k,
  - PRESENT [4] is a block cipher design, i.e., a design for Enc and Dec.

Notes:

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (2)

- Problem: confidential (bulk) communication, per

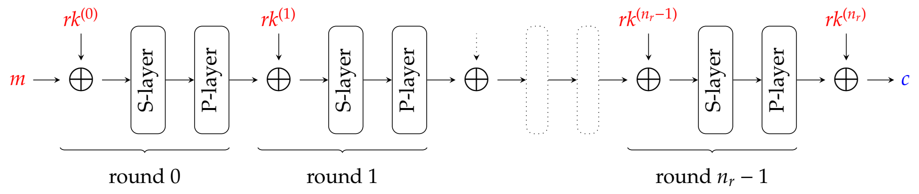


- Solution: **encryption**.
  - one option is to use a **block cipher**, which assumes $\mathcal{H}_i$ and $\mathcal{H}_j$ know k,
  - PRESENT [4] is a block cipher design, i.e., a design for Enc and Dec.

Notes:

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (3)

- The design of (or algorithm for) PRESENT is based on an **SP-network**, i.e.,



where

- each 64-bit round key $rk^{(i)}$ stems from the 80-bit cipher key k,
- m is an 64-bit plaintext block,
- c is an 64-bit ciphertext block,
- there are $n_r = 31$ rounds.

Notes:

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (4)

A constructive case: efficient implementation

- The S-layer is defined as

$$\textsc{S-Layer}: \left\{ \begin{array}{lcl} \{0,1\}^{64} & \rightarrow & \{0,1\}^{64} \\ x & \mapsto & \textsc{S-Box}(x_{63,\ldots,60}) \parallel \cdots \parallel \textsc{S-Box}(x_{7,\ldots,4}) \parallel \textsc{S-Box}(x_{3,\ldots,0}) \end{array} \right.$$

i.e., each 4-bit nibble t in x is substituted by $\textsc{S-Box}(t)$, where

$$\textsc{S-Box}: \left\{ \begin{array}{lcl} \{0,1\}^{4} & \rightarrow & \{0,1\}^{4} \\ x & \mapsto & \left\{ \begin{array}{llll} C_{(16)} & \text{if } x = 0_{(16)} & 3_{(16)} & \text{if } x = 8_{(16)} \\ 5_{(16)} & \text{if } x = 1_{(16)} & E_{(16)} & \text{if } x = 9_{(16)} \\ 6_{(16)} & \text{if } x = 2_{(16)} & F_{(16)} & \text{if } x = A_{(16)} \\ B_{(16)} & \text{if } x = 3_{(16)} & 8_{(16)} & \text{if } x = B_{(16)} \\ 9_{(16)} & \text{if } x = 4_{(16)} & 4_{(16)} & \text{if } x = C_{(16)} \\ 0_{(16)} & \text{if } x = 5_{(16)} & 7_{(16)} & \text{if } x = D_{(16)} \\ A_{(16)} & \text{if } x = 6_{(16)} & 1_{(16)} & \text{if } x = E_{(16)} \\ D_{(16)} & \text{if } x = 7_{(16)} & 2_{(16)} & \text{if } x = F_{(16)} \end{array} \right. \end{array} \right.$$

Notes:

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (4)

A constructive case: efficient implementation

- The S-layer is defined as

$$\textsc{S-Layer}: \left\{ \begin{array}{lcl} \{0,1\}^{64} & \rightarrow & \{0,1\}^{64} \\ x & \mapsto & \textsc{S-Box}(x_{63,\ldots,60}) \parallel \cdots \parallel \textsc{S-Box}(x_{7,\ldots,4}) \parallel \textsc{S-Box}(x_{3,\ldots,0}) \end{array} \right.$$

i.e., each 4-bit nibble t in x is substituted by $\textsc{S-Box}(t)$, where

$$\textsc{S-Box}: \left\{ \begin{array}{lcl} \{0,1\}^{4} & \rightarrow & \{0,1\}^{4} \\ x & \mapsto & \ldots \end{array} \right.$$

x_3	x_2	x_1	x_0	r_3	r_2	r_1	r_0
0	0	0	0	1	1	0	0
0	0	0	1	0	1	0	1
0	0	1	0	0	1	1	0
0	0	1	1	1	0	1	1
0	1	0	0	1	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	1	0	1	0
0	1	1	1	1	1	0	1

x_3	x_2	x_1	x_0	r_3	r_2	r_1	r_0
1	0	0	0	0	0	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	0	1	0	0
1	1	0	1	0	1	1	1
1	1	1	0	0	0	0	1
1	1	1	1	0	0	1	0

Notes:

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (5)
A constructive case: efficient implementation

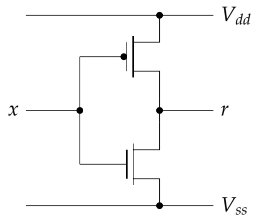
- So what?!
  - PRESENT is a real, standardised, deployed design,
  - the P-layer is a permutation (i.e., rewiring), so there is no computation required,
  - the S-layer is a Boolean function,

  $\therefore$ at this point, we can (more or less) implement PRESENT in hardware.

Notes:

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (6)
A destructive case: information leakage

- We know the power consumption of a CMOS-based NOT gate



  will stem from *two* components, namely

  1. **static consumption**, when transistor states remain fixed (sometimes termed leakage consumption), and
  2. **dynamic consumption**, when transistor states change (resulting in so-called switching activity).

- This means
  - if $x = 0$, setting $x \leftarrow 0$ leads to low power consumption (static only),
  - if $x = 0$, setting $x \leftarrow 1$ leads to high power consumption (static plus dynamic),
  - if $x = 1$, setting $x \leftarrow 0$ leads to high power consumption (static plus dynamic), and
  - if $x = 1$, setting $x \leftarrow 1$ leads to low power consumption (static only).

Notes:

Boolean algebra ⇝ the PRESENT block cipher (7)
A destructive case: information leakage

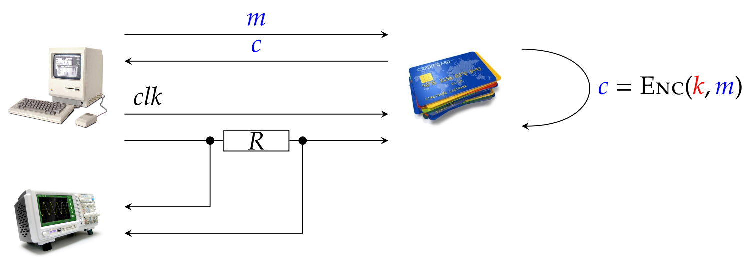
- So what?!
  1. The power consumption of a more general CMOS-based circuit will be
     - *design*-dependent, i.e., depends on the number, type, and connections between transistors, and
     - *data*-dependent, i.e., depends on switching activity of transistors.

Notes:

Boolean algebra ⇝ the PRESENT block cipher (7)
A destructive case: information leakage

- So what?!

  2. Imagine an attacker constructs a mechanism, e.g.,



to acquire a power consumption **trace**, i.e., a sequence

$$s = \langle s_0, s_1, \ldots, s_{l-1} \rangle$$

of l instantaneous samples (here as a result of applying Ohm's law, $V_{dd} = I \cdot R$).

Notes:

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (8)
A destructive case: information leakage

Demo

Notes:

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (9)
A destructive case: information leakage

- Question: how do we *solve* this problem?!
- Answer: see, e.g., [6] or [3, Chapters 7+9], including
  1. **balancing**, and
  2. **masking**.

Notes:

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (10)

A destructive case: information leakage

- (High-level) idea: **balancing**.

Notes:

- We already know enough to define a Boolean function
$$f : \{0,1\} \rightarrow \{0,1\},$$
which maps an n-bit inputs to a 1-bit output. Several properties are important wrt. WDDL:
  - Imagine we have an n-bit input x, and toggle some i-th bit x_i: this means we change x_i from 0 to 1 or from 1 to 0, which we denote $x_i : 0 \rightarrow 1$ and $x_i : 1 \rightarrow 0$. A Boolean function f is described as **monotonic** iff. toggling x_i implies *either* $f(x) : 0 \rightarrow 1$ *or* $f(x) : 1 \rightarrow 0$ (i.e., $f(x)$ also toggles). By inspecting their truth tables, it is clear, for example, that AND is monotonic whereas XOR is non-monotonic.
  - A monotonic Boolean function f is described as **positive monotonic** (resp. **negative monotonic**) if x_i toggles in the same (resp. the opposite) direction as (resp. to) $f(x)$. By inspecting their truth tables, it is clear, for example, that OR is positive monotonic whereas NAND is negative monotonic.
  - Positive monotonic functions are st. if all $x_i = 0$ then $f(x) = 0$. If this were *not* the case, a toggle $x_i : 0 \rightarrow 1$ could not provoke $f(x) : 0 \rightarrow 1$ (because $f(x) = 1$ already). The opposite is also true, i.e. if all $x_i = 1$ then $f(x) = 1$.
- WDDL requires f_0 and f_1 are positive monotonic, and, in a general sense, are defined st.
$$f_0(x_0, x_1, \ldots, x_{n-1}) \equiv \neg f_1(\neg x_0, \neg x_1, \ldots, \neg x_{n-1}).$$
- WDDL aims to ensure that whatever it is currently computing or has previously computed, the cell *always* produces the same switching behaviour (i.e., the same number of toggles). It does this by considering a 2-phase (vs. a genuine, continuous or combinatorial) approach to computation:
  1. Phase #1 is termed **evaluation**.
     - The previous pre-charge phase will have set all $x_i = 0$, so, due to their positive monotonicity, it *must* be the case that $f_0(x) = 0$ and $f_1(x) = 0$.
     - When inputs to f_0 and f_1 are set to the intended value, those inputs *only* toggle $0 \rightarrow 1$.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $0 \rightarrow 1$ and the other remains 0.
  2. Phase #2 is termed **pre-charge**.
     - Each x_i that *was* 1 is set to 0: this means we must have $x_i : 1 \rightarrow 0$ in each case.
     - Doing so means *if* the output from f_0 or f_1 toggles, then, due to their positive monotonicity, it *must* be the case that $f_0(x) : 1 \rightarrow 0$ *or* $f_1(x) : 1 \rightarrow 0$ as well.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $1 \rightarrow 0$ and the other remains 0.

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (10)

A destructive case: information leakage

- (High-level) idea: **balancing**.
  1. update the wires, e.g., via dual-rail (vs. single-rail):
     - represent each wire as *two*, separate wires st.
$$x \mapsto \hat{x} = (\hat{x}_0, \hat{x}_1) = (x, \neg x) = \begin{cases} (0,1) & \text{if } x = 0 \\ (1,0) & \text{if } x = 1 \end{cases},$$
     - now the representation of x has *constant* weight (i.e., we can't distinguish $x = 0$ from $x = 1$), *plus*
     - we don't need any NOT gates!

Notes:

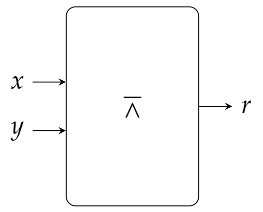
- We already know enough to define a Boolean function
$$f : \{0,1\} \rightarrow \{0,1\},$$
which maps an n-bit inputs to a 1-bit output. Several properties are important wrt. WDDL:
  - Imagine we have an n-bit input x, and toggle some i-th bit x_i: this means we change x_i from 0 to 1 or from 1 to 0, which we denote $x_i : 0 \rightarrow 1$ and $x_i : 1 \rightarrow 0$. A Boolean function f is described as **monotonic** iff. toggling x_i implies *either* $f(x) : 0 \rightarrow 1$ *or* $f(x) : 1 \rightarrow 0$ (i.e., $f(x)$ also toggles). By inspecting their truth tables, it is clear, for example, that AND is monotonic whereas XOR is non-monotonic.
  - A monotonic Boolean function f is described as **positive monotonic** (resp. **negative monotonic**) if x_i toggles in the same (resp. the opposite) direction as (resp. to) $f(x)$. By inspecting their truth tables, it is clear, for example, that OR is positive monotonic whereas NAND is negative monotonic.
  - Positive monotonic functions are st. if all $x_i = 0$ then $f(x) = 0$. If this were *not* the case, a toggle $x_i : 0 \rightarrow 1$ could not provoke $f(x) : 0 \rightarrow 1$ (because $f(x) = 1$ already). The opposite is also true, i.e. if all $x_i = 1$ then $f(x) = 1$.
- WDDL requires f_0 and f_1 are positive monotonic, and, in a general sense, are defined st.
$$f_0(x_0, x_1, \ldots, x_{n-1}) \equiv \neg f_1(\neg x_0, \neg x_1, \ldots, \neg x_{n-1}).$$
- WDDL aims to ensure that whatever it is currently computing or has previously computed, the cell *always* produces the same switching behaviour (i.e., the same number of toggles). It does this by considering a 2-phase (vs. a genuine, continuous or combinatorial) approach to computation:
  1. Phase #1 is termed **evaluation**.
     - The previous pre-charge phase will have set all $x_i = 0$, so, due to their positive monotonicity, it *must* be the case that $f_0(x) = 0$ and $f_1(x) = 0$.
     - When inputs to f_0 and f_1 are set to the intended value, those inputs *only* toggle $0 \rightarrow 1$.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $0 \rightarrow 1$ and the other remains 0.
  2. Phase #2 is termed **pre-charge**.
     - Each x_i that *was* 1 is set to 0: this means we must have $x_i : 1 \rightarrow 0$ in each case.
     - Doing so means *if* the output from f_0 or f_1 toggles, then, due to their positive monotonicity, it *must* be the case that $f_0(x) : 1 \rightarrow 0$ *or* $f_1(x) : 1 \rightarrow 0$ as well.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $1 \rightarrow 0$ and the other remains 0.

Boolean algebra ⇝ the PRESENT block cipher (10)

A destructive case: information leakage

- (High-level) idea: **balancing**.

2. update the cells, e.g., via Wave Dynamic Differential Logic (WDDL) [8]:



Notes:

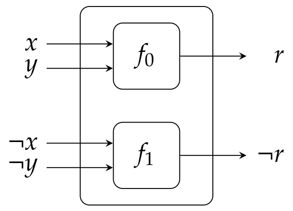
- We already know enough to define a Boolean function
$$f : \{0,1\} \rightarrow \{0,1\},$$
which maps an n-bit inputs to a 1-bit output. Several properties are important wrt. WDDL:
  - Imagine we have an n-bit input x, and toggle some i-th bit x_i: this means we change x_i from 0 to 1 or from 1 to 0, which we denote $x_i : 0 \rightarrow 1$ and $x_i : 1 \rightarrow 0$. A Boolean function f is described as **monotonic** iff. toggling x_i implies *either* $f(x) : 0 \rightarrow 1$ *or* $f(x) : 1 \rightarrow 0$ (i.e., $f(x)$ also toggles). By inspecting their truth tables, it is clear, for example, that AND is monotonic whereas XOR is non-monotonic.
  - A monotonic Boolean function f is described as **positive monotonic** (resp. **negative monotonic**) if x_i toggles in the same (resp. the opposite) direction as (resp. to) $f(x)$. By inspecting their truth tables, it is clear, for example, that OR is positive monotonic whereas NAND is negative monotonic.
  - Positive monotonic functions are st. if all $x_i = 0$ then $f(x) = 0$. If this were *not* the case, a toggle $x_i : 0 \rightarrow 1$ could not provoke $f(x) : 0 \rightarrow 1$ (because $f(x) = 1$ already). The opposite is also true, i.e. if all $x_i = 1$ then $f(x) = 1$.
- WDDL requires f_0 and f_1 are positive monotonic, and, in a general sense, are defined st.
$$f_0(x_0, x_1, \ldots, x_{n-1}) \equiv \neg f_1(\neg x_0, \neg x_1, \ldots, \neg x_{n-1}).$$
- WDDL aims to ensure that whatever it is currently computing or has previously computed, the cell *always* produces the same switching behaviour (i.e., the same number of toggles). It does this by considering a 2-phase (vs. a genuine, continuous or combinatorial) approach to computation:
  1. Phase #1 is termed **evaluation**.
     - The previous pre-charge phase will have set all $x_i = 0$, so, due to their positive monotonicity, it *must* be the case that $f_0(x) = 0$ and $f_1(x) = 0$.
     - When inputs to f_0 and f_1 are set to the intended value, those inputs *only* toggle $0 \rightarrow 1$.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $0 \rightarrow 1$ and the other remains 0.
  2. Phase #2 is termed **pre-charge**.
     - Each x_i that *was* 1 is set to 0: this means we must have $x_i : 1 \rightarrow 0$ in each case.
     - Doing so means *if* the output from f_0 or f_1 toggles, then, due to their positive monotonicity, it *must* be the case that $f_0(x) : 1 \rightarrow 0$ *or* $f_1(x) : 1 \rightarrow 0$ as well.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $1 \rightarrow 0$ and the other remains 0.

Boolean algebra ⇝ the PRESENT block cipher (10)

A destructive case: information leakage

- (High-level) idea: **balancing**.

2. update the cells, e.g., via Wave Dynamic Differential Logic (WDDL) [8]:



Notes:

- We already know enough to define a Boolean function
$$f : \{0,1\} \rightarrow \{0,1\},$$
which maps an n-bit inputs to a 1-bit output. Several properties are important wrt. WDDL:
  - Imagine we have an n-bit input x, and toggle some i-th bit x_i: this means we change x_i from 0 to 1 or from 1 to 0, which we denote $x_i : 0 \rightarrow 1$ and $x_i : 1 \rightarrow 0$. A Boolean function f is described as **monotonic** iff. toggling x_i implies *either* $f(x) : 0 \rightarrow 1$ *or* $f(x) : 1 \rightarrow 0$ (i.e., $f(x)$ also toggles). By inspecting their truth tables, it is clear, for example, that AND is monotonic whereas XOR is non-monotonic.
  - A monotonic Boolean function f is described as **positive monotonic** (resp. **negative monotonic**) if x_i toggles in the same (resp. the opposite) direction as (resp. to) $f(x)$. By inspecting their truth tables, it is clear, for example, that OR is positive monotonic whereas NAND is negative monotonic.
  - Positive monotonic functions are st. if all $x_i = 0$ then $f(x) = 0$. If this were *not* the case, a toggle $x_i : 0 \rightarrow 1$ could not provoke $f(x) : 0 \rightarrow 1$ (because $f(x) = 1$ already). The opposite is also true, i.e. if all $x_i = 1$ then $f(x) = 1$.
- WDDL requires f_0 and f_1 are positive monotonic, and, in a general sense, are defined st.
$$f_0(x_0, x_1, \ldots, x_{n-1}) \equiv \neg f_1(\neg x_0, \neg x_1, \ldots, \neg x_{n-1}).$$
- WDDL aims to ensure that whatever it is currently computing or has previously computed, the cell *always* produces the same switching behaviour (i.e., the same number of toggles). It does this by considering a 2-phase (vs. a genuine, continuous or combinatorial) approach to computation:
  1. Phase #1 is termed **evaluation**.
     - The previous pre-charge phase will have set all $x_i = 0$, so, due to their positive monotonicity, it *must* be the case that $f_0(x) = 0$ and $f_1(x) = 0$.
     - When inputs to f_0 and f_1 are set to the intended value, those inputs *only* toggle $0 \rightarrow 1$.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $0 \rightarrow 1$ and the other remains 0.
  2. Phase #2 is termed **pre-charge**.
     - Each x_i that *was* 1 is set to 0: this means we must have $x_i : 1 \rightarrow 0$ in each case.
     - Doing so means *if* the output from f_0 or f_1 toggles, then, due to their positive monotonicity, it *must* be the case that $f_0(x) : 1 \rightarrow 0$ *or* $f_1(x) : 1 \rightarrow 0$ as well.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $1 \rightarrow 0$ and the other remains 0.

Boolean algebra ⇝ the PRESENT block cipher (10)

A destructive case: information leakage

- (High-level) idea: **balancing**.

2. update the cells, e.g., via Wave Dynamic Differential Logic (WDDL) [8]:

x	y	$\neg x$	$\neg y$	$r = \neg x \vee \neg y$	$\neg r = x \wedge y$
0	0	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	0	1

Notes:

- We already know enough to define a Boolean function

$$f : \{0,1\} \rightarrow \{0,1\},$$

  which maps an n-bit inputs to a 1-bit output. Several properties are important wrt. WDDL:

  - Imagine we have an n-bit input x, and toggle some i-th bit x_i: this means we change x_i from 0 to 1 or from 1 to 0, which we denote $x_i : 0 \rightarrow 1$ and $x_i : 1 \rightarrow 0$. A Boolean function f is described as **monotonic** iff. toggling x_i implies *either* $f(x) : 0 \rightarrow 1$ *or* $f(x) : 1 \rightarrow 0$ (i.e., $f(x)$ also toggles). By inspecting their truth tables, it is clear, for example, that AND is monotonic whereas XOR is non-monotonic.
  - A monotonic Boolean function f is described as **positive monotonic** (resp. **negative monotonic**) if x_i toggles in the same (resp. the opposite) direction as (resp. to) $f(x)$. By inspecting their truth tables, it is clear, for example, that OR is positive monotonic whereas NAND is negative monotonic.
  - Positive monotonic functions are st. if all $x_i = 0$ then $f(x) = 0$. If this were *not* the case, a toggle $x_i : 0 \rightarrow 1$ could not provoke $f(x) : 0 \rightarrow 1$ (because $f(x) = 1$ already). The opposite is also true, i.e. if all $x_i = 1$ then $f(x) = 1$.

- WDDL requires f_0 and f_1 are positive monotonic, and, in a general sense, are defined st.

$$f_0(x_0, x_1, \ldots, x_{n-1}) \equiv \neg f_1(\neg x_0, \neg x_1, \ldots, \neg x_{n-1}).$$

- WDDL aims to ensure that whatever it is currently computing or has previously computed, the cell *always* produces the same switching behaviour (i.e., the same number of toggles). It does this by considering a 2-phase (vs. a genuine, continuous or combinatorial) approach to computation:

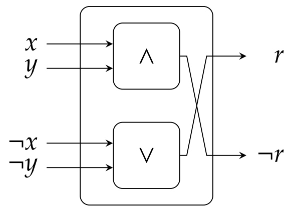
  1. Phase #1 is termed **evaluation**.
     - The previous pre-charge phase will have set all $x_i = 0$, so, due to their positive monotonicity, it *must* be the case that $f_0(x) = 0$ and $f_1(x) = 0$.
     - When inputs to f_0 and f_1 are set to the intended value, those inputs *only* toggle $0 \rightarrow 1$.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $0 \rightarrow 1$ and the other remains 0.
  2. Phase #2 is termed **pre-charge**.
     - Each x_i that *was* 1 is set to 0: this means we must have $x_i : 1 \rightarrow 0$ in each case.
     - Doing so means *if* the output from f_0 or f_1 toggles, then, due to their positive monotonicity, it *must* be the case that $f_0(x) : 1 \rightarrow 0$ *or* $f_1(x) : 1 \rightarrow 0$ as well.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $1 \rightarrow 0$ and the other remains 0.

Boolean algebra ⇝ the PRESENT block cipher (10)

A destructive case: information leakage

- (High-level) idea: **balancing**.

2. update the cells, e.g., via Wave Dynamic Differential Logic (WDDL) [8]:



Notes:

- We already know enough to define a Boolean function

$$f : \{0,1\} \rightarrow \{0,1\},$$

  which maps an n-bit inputs to a 1-bit output. Several properties are important wrt. WDDL:

  - Imagine we have an n-bit input x, and toggle some i-th bit x_i: this means we change x_i from 0 to 1 or from 1 to 0, which we denote $x_i : 0 \rightarrow 1$ and $x_i : 1 \rightarrow 0$. A Boolean function f is described as **monotonic** iff. toggling x_i implies *either* $f(x) : 0 \rightarrow 1$ *or* $f(x) : 1 \rightarrow 0$ (i.e., $f(x)$ also toggles). By inspecting their truth tables, it is clear, for example, that AND is monotonic whereas XOR is non-monotonic.
  - A monotonic Boolean function f is described as **positive monotonic** (resp. **negative monotonic**) if x_i toggles in the same (resp. the opposite) direction as (resp. to) $f(x)$. By inspecting their truth tables, it is clear, for example, that OR is positive monotonic whereas NAND is negative monotonic.
  - Positive monotonic functions are st. if all $x_i = 0$ then $f(x) = 0$. If this were *not* the case, a toggle $x_i : 0 \rightarrow 1$ could not provoke $f(x) : 0 \rightarrow 1$ (because $f(x) = 1$ already). The opposite is also true, i.e. if all $x_i = 1$ then $f(x) = 1$.

- WDDL requires f_0 and f_1 are positive monotonic, and, in a general sense, are defined st.

$$f_0(x_0, x_1, \ldots, x_{n-1}) \equiv \neg f_1(\neg x_0, \neg x_1, \ldots, \neg x_{n-1}).$$

- WDDL aims to ensure that whatever it is currently computing or has previously computed, the cell *always* produces the same switching behaviour (i.e., the same number of toggles). It does this by considering a 2-phase (vs. a genuine, continuous or combinatorial) approach to computation:

  1. Phase #1 is termed **evaluation**.
     - The previous pre-charge phase will have set all $x_i = 0$, so, due to their positive monotonicity, it *must* be the case that $f_0(x) = 0$ and $f_1(x) = 0$.
     - When inputs to f_0 and f_1 are set to the intended value, those inputs *only* toggle $0 \rightarrow 1$.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $0 \rightarrow 1$ and the other remains 0.
  2. Phase #2 is termed **pre-charge**.
     - Each x_i that *was* 1 is set to 0: this means we must have $x_i : 1 \rightarrow 0$ in each case.
     - Doing so means *if* the output from f_0 or f_1 toggles, then, due to their positive monotonicity, it *must* be the case that $f_0(x) : 1 \rightarrow 0$ *or* $f_1(x) : 1 \rightarrow 0$ as well.
     - Since they are complimentary, one of $f_0(x)$ and $f_1(x)$ always toggles $1 \rightarrow 0$ and the other remains 0.

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (11)

A destructive case: information leakage

- (High-level) idea: **masking**.

Notes:

- Although note stated explicitly on the slide, the idea is that
  - vs. the general-form masked cell, MDPL uses a) a *single* mask m for *all* wires, and b) a dual-rail representation for those wires,
  - the MDPL cell is then similar to WDDL: f_0 and f_1 are basically just replaced by the majority function,
  - operation of the cell is also similar therefore, with both an evaluation and pre-charge phase required.

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (11)

A destructive case: information leakage

- (High-level) idea: **masking**.
  1. update the wires, e.g., via a Boolean (vs. arithmetic) mask:
     - select a random mask m,
     - represent each wire as

$$x \mapsto \hat{x}_m = x \oplus m,$$

     - now the representation of x has *random* weight (i.e., we can't distinguish $x = 0$, $m = 1$ from $x = 1$, $m = 0$).

Notes:

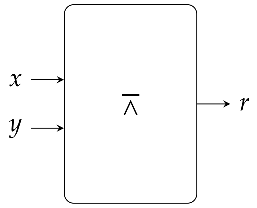
- Although note stated explicitly on the slide, the idea is that
  - vs. the general-form masked cell, MDPL uses a) a *single* mask m for *all* wires, and b) a dual-rail representation for those wires,
  - the MDPL cell is then similar to WDDL: f_0 and f_1 are basically just replaced by the majority function,
  - operation of the cell is also similar therefore, with both an evaluation and pre-charge phase required.

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (11)

A destructive case: information leakage

- (High-level) idea: **masking**.

2. update the cells, e.g., via Masked Dual-rail Pre-charge Logic (MDPL) [7].



Notes:

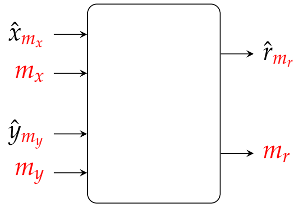
- Although note stated explicitly on the slide, the idea is that
  - vs. the general-form masked cell, MDPL uses a) a *single* mask m for *all* wires, and b) a dual-rail representation for those wires,
  - the MDPL cell is then similar to WDDL: f_0 and f_1 are basically just replaced by the majority function,
  - operation of the cell is also similar therefore, with both an evaluation and pre-charge phase required.

Boolean algebra $\rightsquigarrow$ the PRESENT block cipher (11)

A destructive case: information leakage

- (High-level) idea: **masking**.

2. update the cells, e.g., via Masked Dual-rail Pre-charge Logic (MDPL) [7].



Notes:

- Although note stated explicitly on the slide, the idea is that
  - vs. the general-form masked cell, MDPL uses a) a *single* mask m for *all* wires, and b) a dual-rail representation for those wires,
  - the MDPL cell is then similar to WDDL: f_0 and f_1 are basically just replaced by the majority function,
  - operation of the cell is also similar therefore, with both an evaluation and pre-charge phase required.

Boolean algebra ⇝ the PRESENT block cipher (11)

A destructive case: information leakage

- (High-level) idea: **masking**.

2. update the cells, e.g., via Masked Dual-rail Pre-charge Logic (MDPL) [7].

x	y	z	$\textsc{Maj}(x,y,z)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Notes:

- Although note stated explicitly on the slide, the idea is that
  - vs. the general-form masked cell, MDPL uses a) a *single* mask m for *all* wires, and b) a dual-rail representation for those wires,
  - the MDPL cell is then similar to WDDL: f_0 and f_1 are basically just replaced by the majority function,
  - operation of the cell is also similar therefore, with both an evaluation and pre-charge phase required.

Boolean algebra ⇝ the PRESENT block cipher (11)

A destructive case: information leakage

- (High-level) idea: **masking**.

2. update the cells, e.g., via Masked Dual-rail Pre-charge Logic (MDPL) [7].

$\hat{x}_m$	$\hat{y}_m$	m	x	y	$\neg\hat{x}_m$	$\neg\hat{y}_m$	$\neg m$	$\hat{r}_m = \textsc{Maj}(\neg\hat{x}_m, \neg\hat{y}_m, \neg m)$	$\neg\hat{r}_m = \textsc{Maj}(\hat{x}_m, \hat{y}_m, m)$	r
0	0	0	0	0	1	1	1	1	0	1
0	0	1	1	1	1	1	0	1	0	0
0	1	0	0	1	1	0	1	1	0	1
0	1	1	1	0	1	0	0	0	1	1
1	0	0	1	0	0	1	1	1	0	1
1	0	1	0	1	0	1	0	0	1	1
1	1	0	1	1	0	0	1	0	1	0
1	1	1	0	0	0	0	0	0	1	1

Notes:

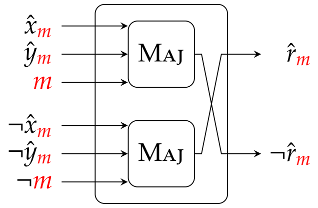
- Although note stated explicitly on the slide, the idea is that
  - vs. the general-form masked cell, MDPL uses a) a *single* mask m for *all* wires, and b) a dual-rail representation for those wires,
  - the MDPL cell is then similar to WDDL: f_0 and f_1 are basically just replaced by the majority function,
  - operation of the cell is also similar therefore, with both an evaluation and pre-charge phase required.

Boolean algebra ⇝ the PRESENT block cipher (11)

A destructive case: information leakage

- (High-level) idea: **masking**.

2. update the cells, e.g., via Masked Dual-rail Pre-charge Logic (MDPL) [7].



Notes:

- Although note stated explicitly on the slide, the idea is that
  - vs. the general-form masked cell, MDPL uses a) a *single* mask m for *all* wires, and b) a dual-rail representation for those wires,
  - the MDPL cell is then similar to WDDL: f_0 and f_1 are basically just replaced by the majority function,
  - operation of the cell is also similar therefore, with both an evaluation and pre-charge phase required.

Conclusions

- Take away points: in a general sense, keep in mind that
1. you can already solve real problems, e.g.,
   - implement a given design,
   - reason about properties of a given design wrt. suitable metrics,
   - ...
2. with a deep(er) understanding, e.g.,
   - reasoning about representation of data,
   - reasoning about and manipulating forms of computation,
   - ...

   (more) interesting and innovative solutions become viable.

Notes:

Additional Reading

- *Wikipedia: Block cipher*. URL: http://en.wikipedia.org/wiki/Block_cipher.
- *Wikipedia: Side-Channel Attack*. URL: http://en.wikipedia.org/wiki/Side_channel_attack.
- P.C. Kocher et al. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering* 1.1 (2011), pp. 5–27.

Notes:

References

[1] *Wikipedia: Block cipher*. URL: http://en.wikipedia.org/wiki/Block_cipher (see p. 65).

[2] *Wikipedia: Side-Channel Attack*. URL: http://en.wikipedia.org/wiki/Side_channel_attack (see p. 65).

[3] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007 (see p. 35).

[4] A. Bogdanov et al. "PRESENT: An Ultra-Lightweight Block Cipher". In: *Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag LNCS 4727, 2007, pp. 450–466 (see pp. 7, 9, 11, 13, 15, 17).

[5] P.C. Kocher et al. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering* 1.1 (2011), pp. 5–27 (see p. 65).

[6] M. Mayhew and R. Muresan. "An overview of hardware-level statistical power analysis attack countermeasures". In: *Journal of Cryptographic Engineering* (2016), to appear (see p. 35).

[7] T. Popp and S. Mangard. "Masked Dual-rail Pre-charge Logic: DPA-resistance Without Routing Constraints". In: *Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag LNCS 3659, 2005, pp. 172–186 (see pp. 49, 51, 53, 55, 57, 59, 61).

[8] K. Tiri and I. Verbauwhede. "A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation". In: *Design, Automation and Test in Europe (DATE)*. 2004, 1:246–1:251 (see pp. 37, 39, 41, 43, 45, 47).

Notes: