Avoiding Leaks in Hardware and Software

My Journey Towards Full-Stack Security

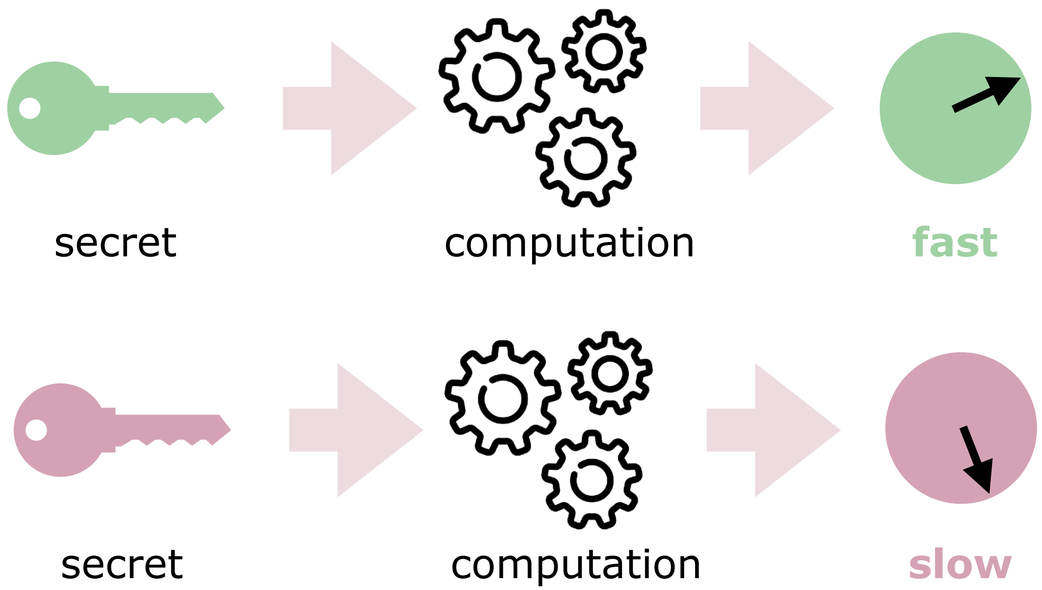
Klaus v. Gleissenthall

Assistant Professor

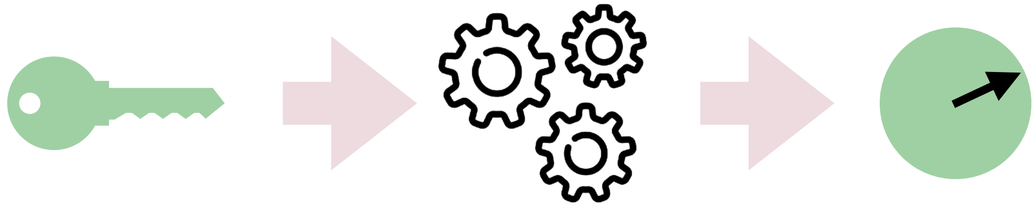
Computers know our **secrets**,
but will they keep them **safe**?

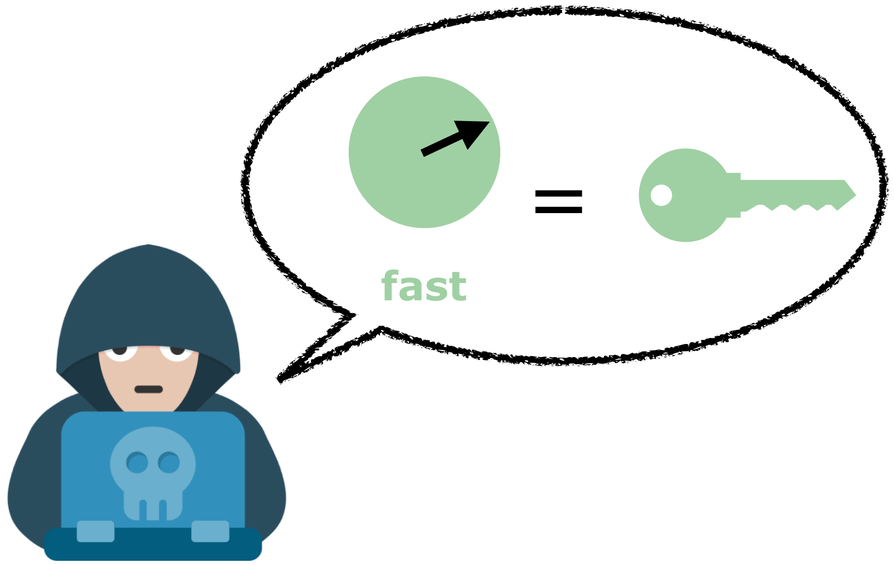
Side-channels leak secrets **indirectly**

Timing Leaks

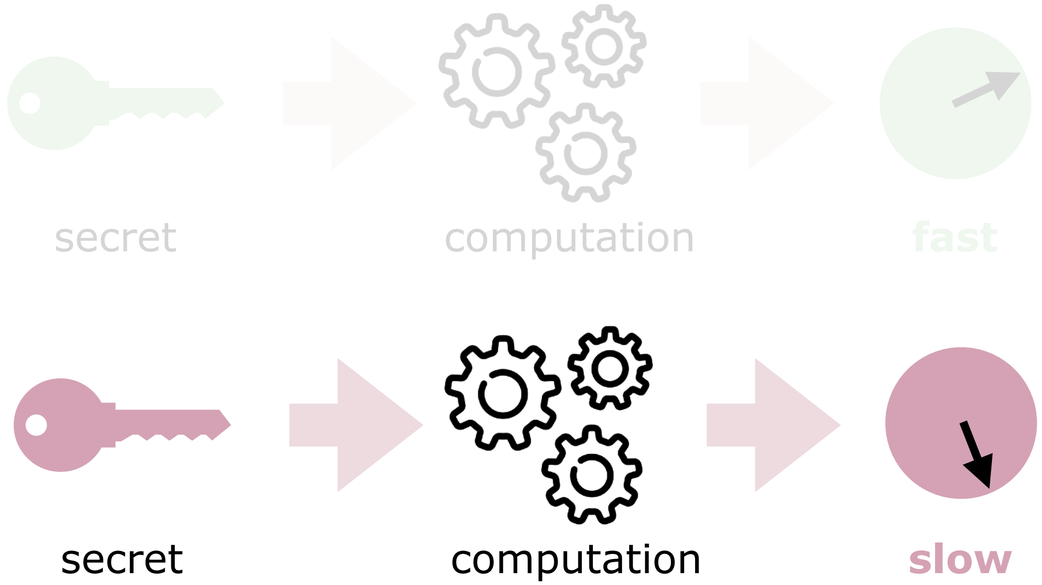


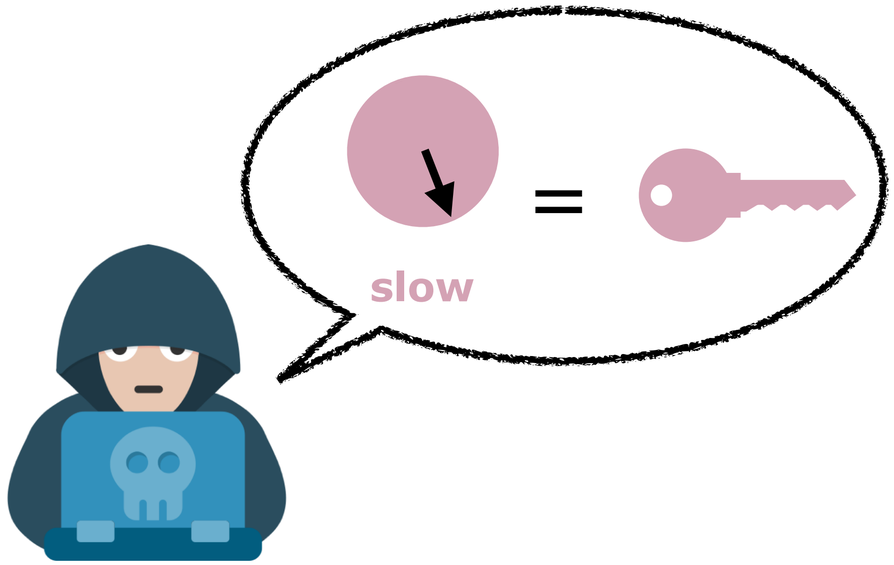
Timing Leaks



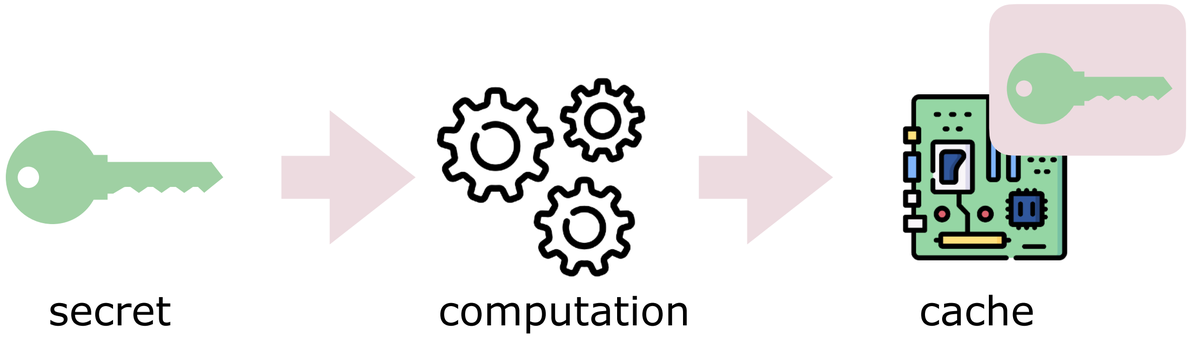


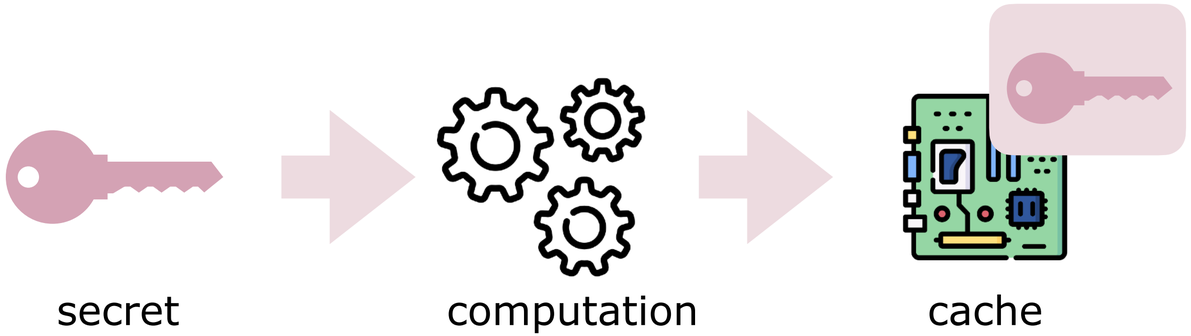
Timing Leaks



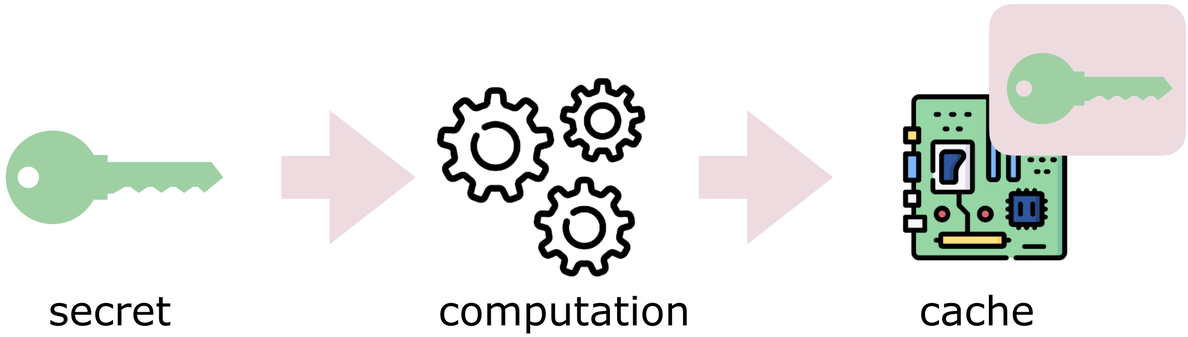


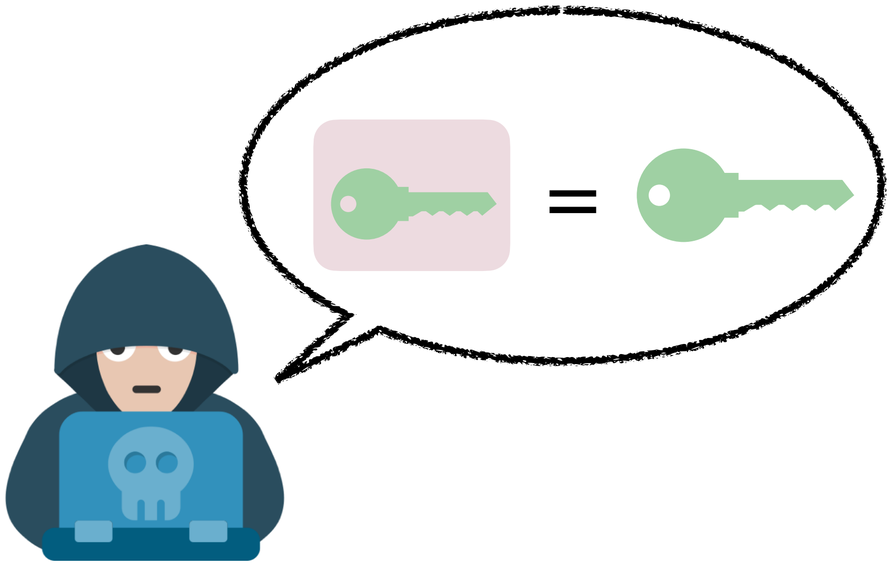
Cache Leaks



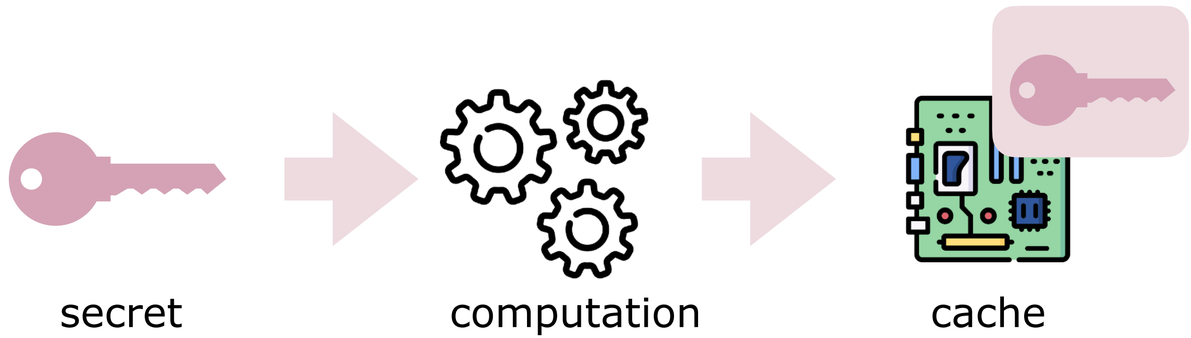


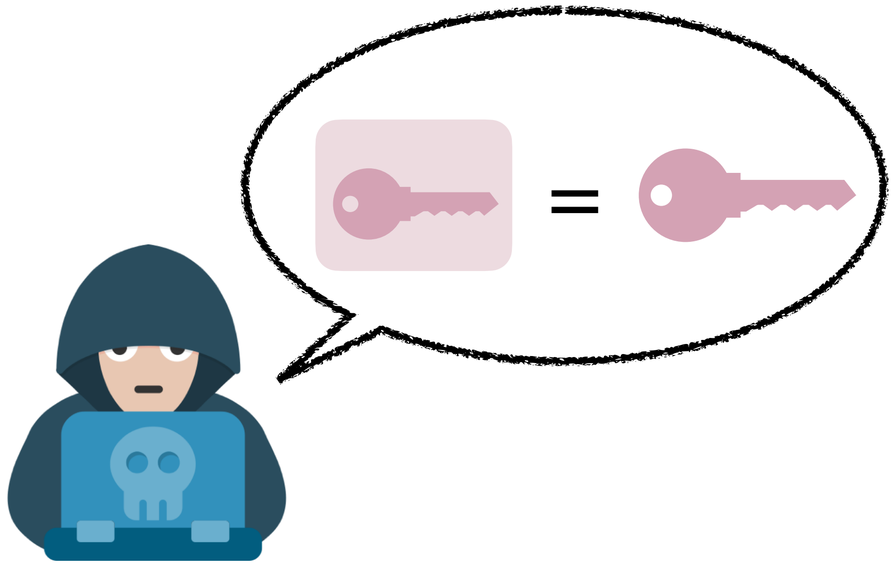
Cache Leaks





Cache Leaks





Side-Channels

Used to break:

 RSA

 DB

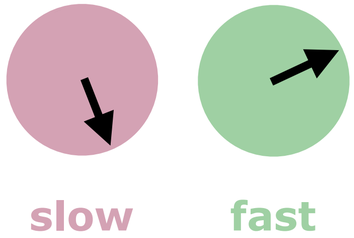
 Neural Nets

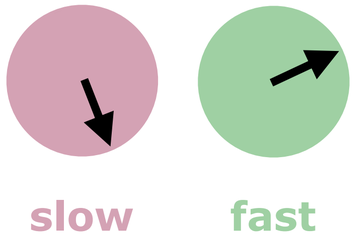
Programming Model

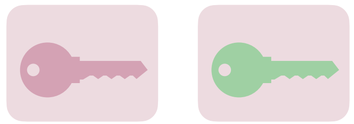
(1) `div` secret

(2) `if` secret `{…}`

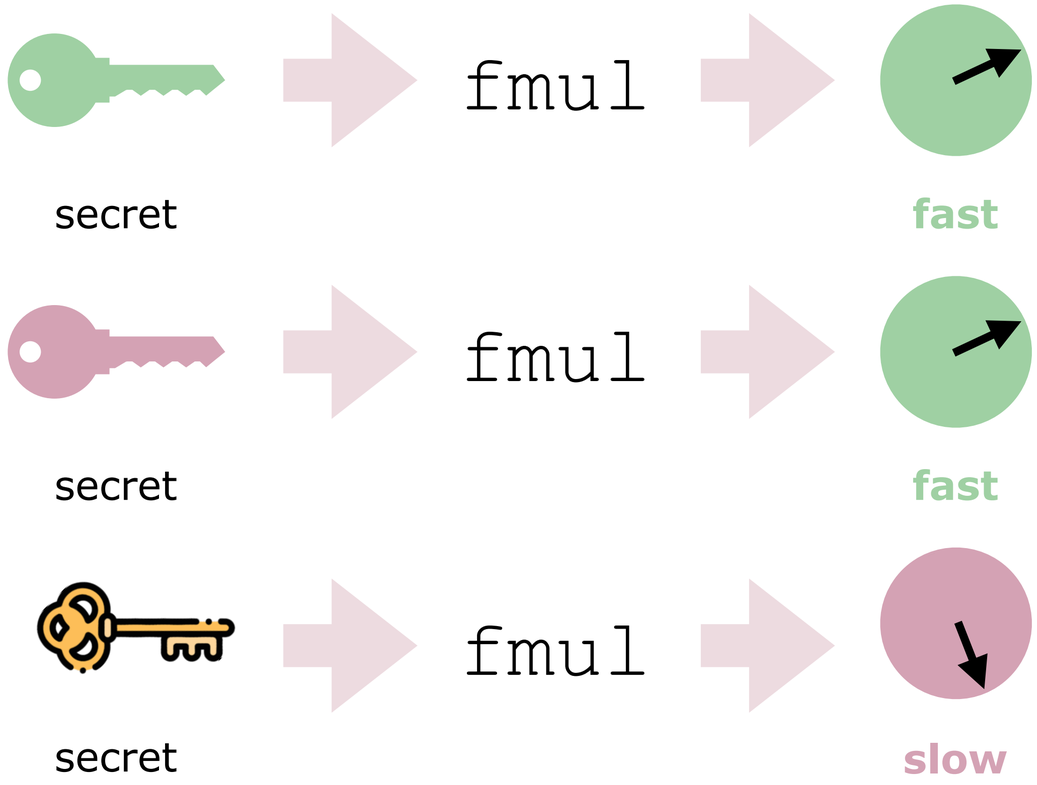
(3) `a[` secret `]`

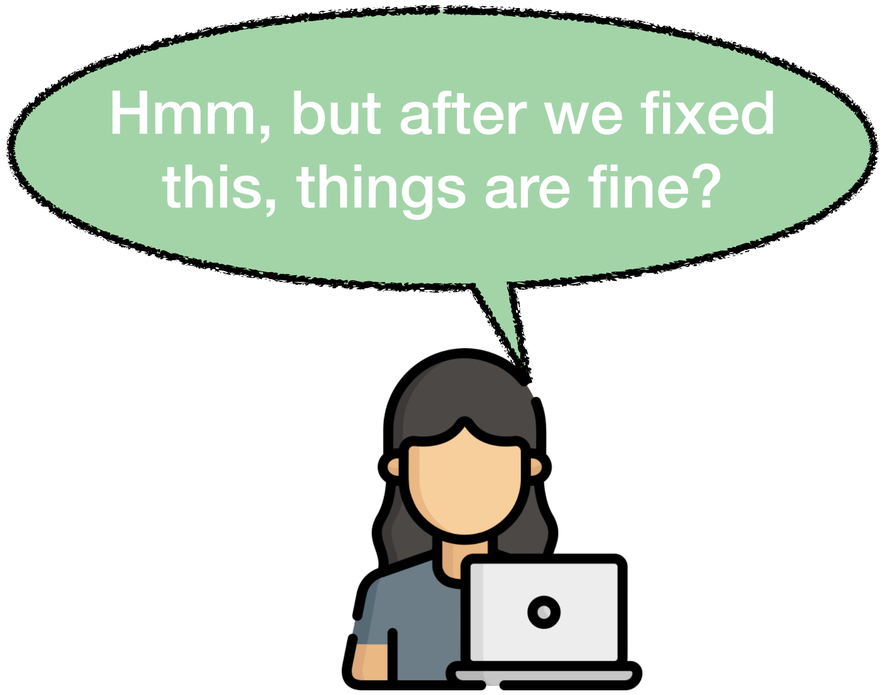




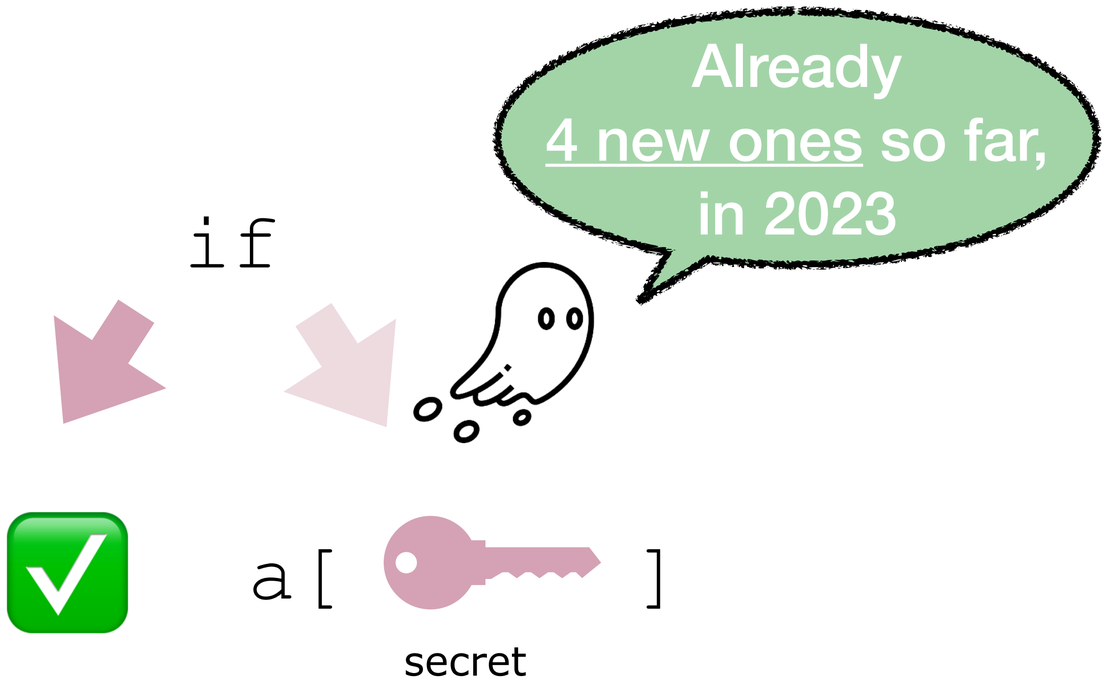


Past Failures

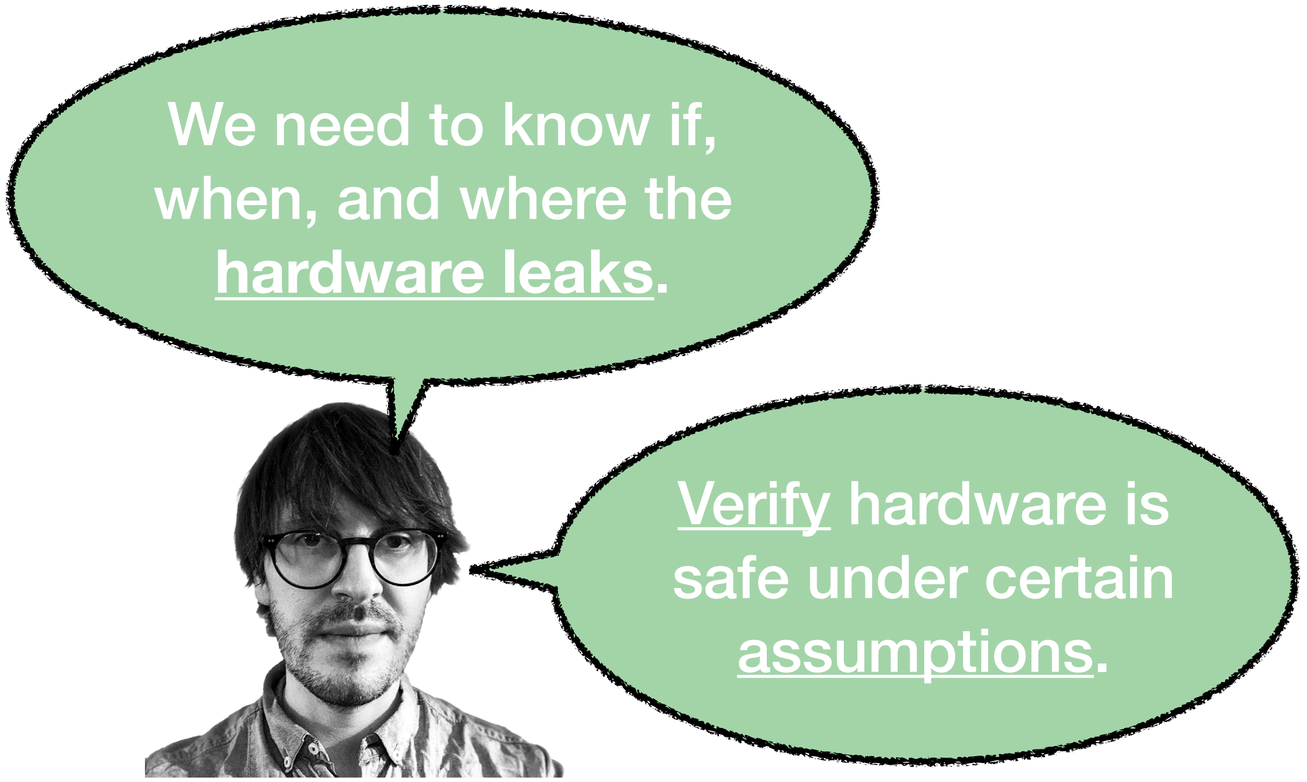




Past Failures



My Main Idea


We need to know if, when, and where the hardware leaks.
Verify hardware is safe under certain assumptions.

My Journey

[1] Iodine: What's timing in HW. **Usenix Security**'19.

[2] Xenon: Help non-experts find assumptions. **CCS**'21.

[3] LeaVe: Assumptions at Software level. **CCS**'23.
Distinguished paper.

My Journey

[1] Iodine: What's timing in HW. **Usenix Security**'19.

[2] Xenon: Help non-experts find assumptions. **CCS**'21.

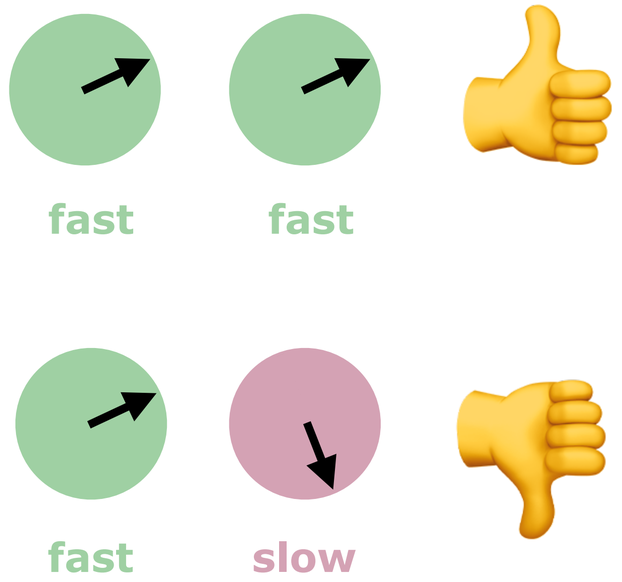
[3] LeaVe: Assumptions at Software level. **CCS**'23.
Distinguished paper.

Iodine: Timing in Hardware

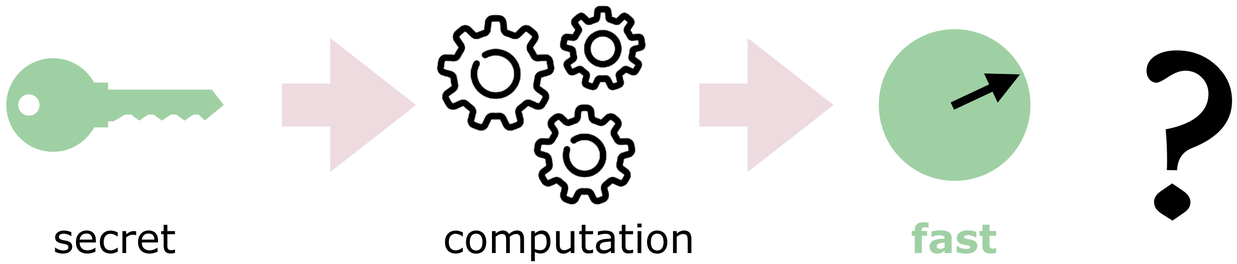
Verilog

Iodine

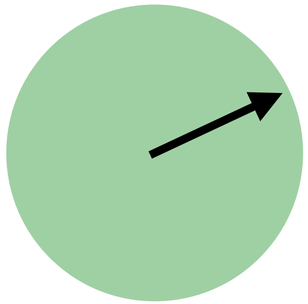
constant time



Problem: What does *time* mean in hardware?



Problem: What does ***time*** mean in hardware?



?

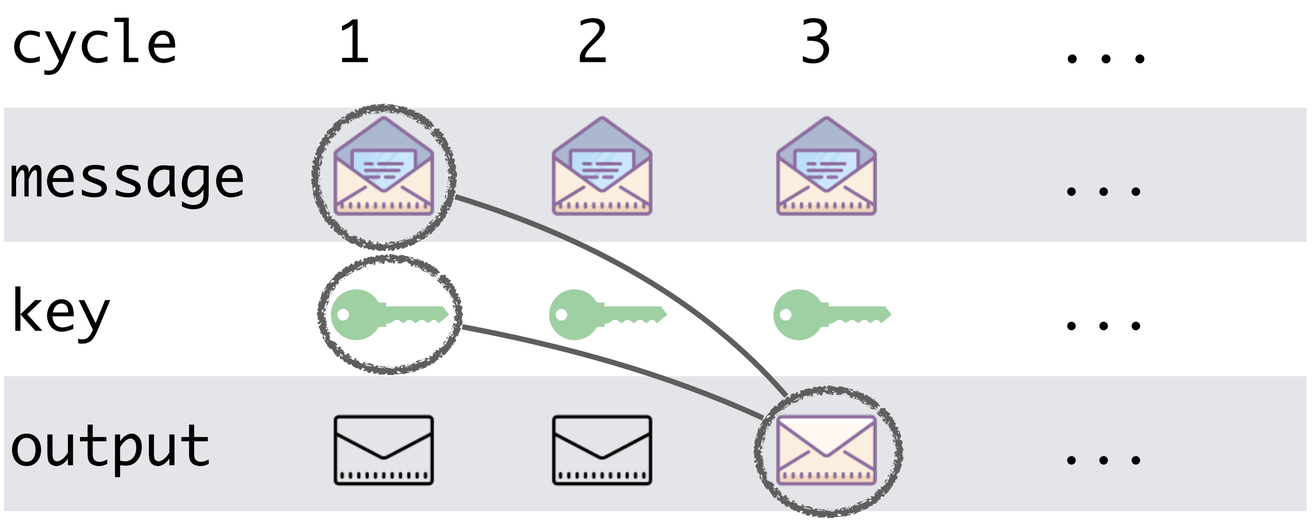
Input/Output Stream

Inputs over Multiple Cycles

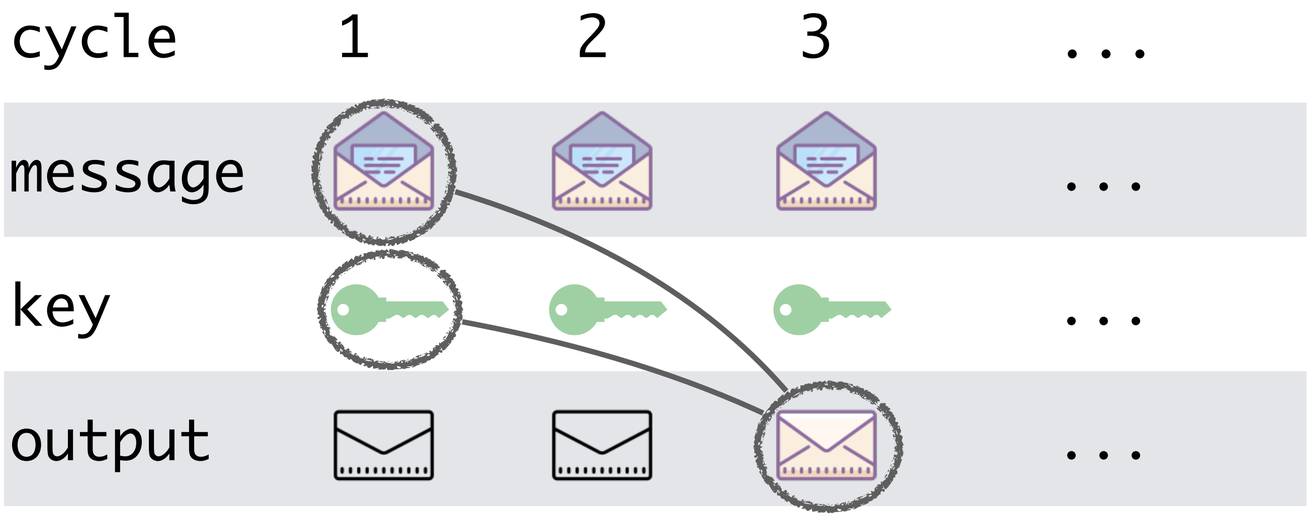
Outputs over Multiple Cycles

Problem: What does *time* mean in hardware?

Problem: Input/Output Stream

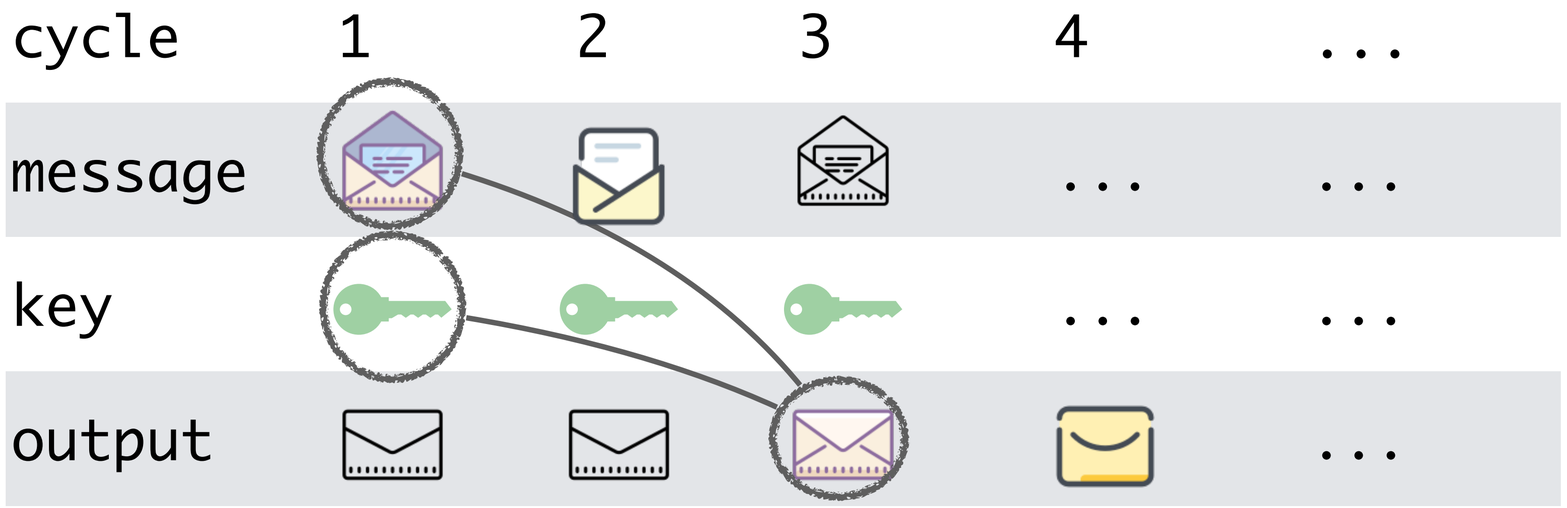


Problem: Input/Output Stream



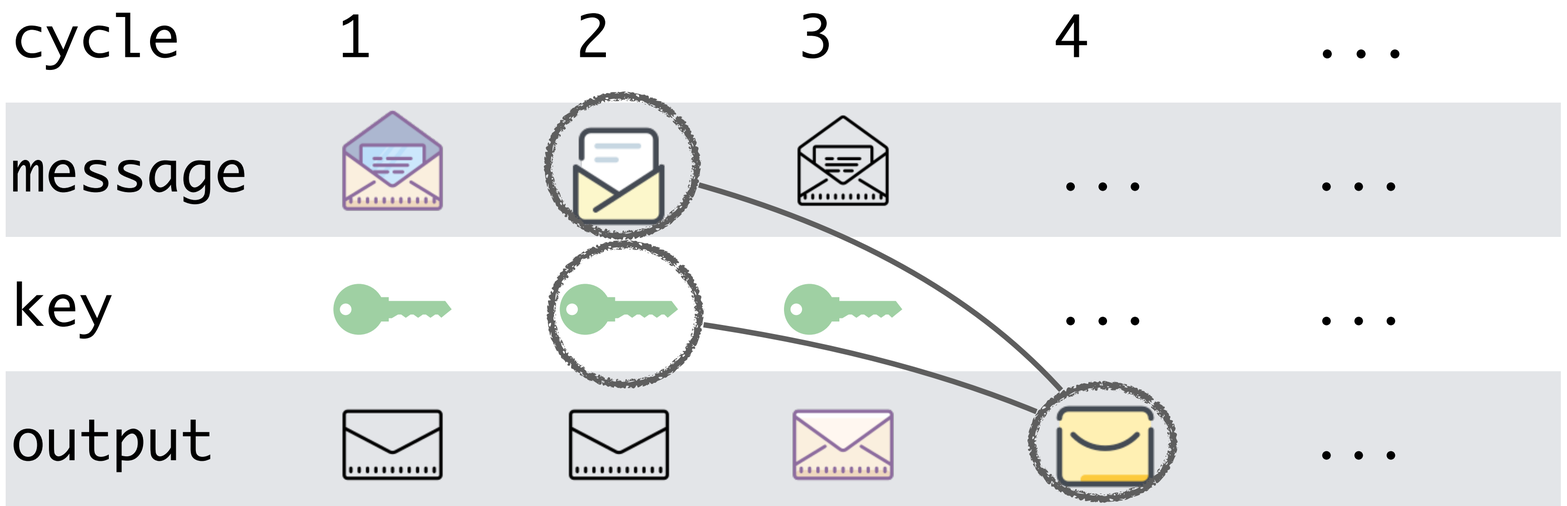
Which input belongs to which output?

Problem: Input/Output Stream & Pipelining



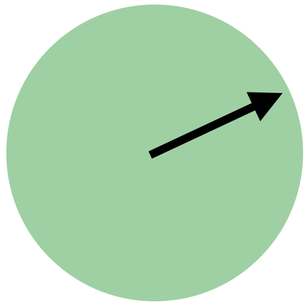
Which input belongs to which output?

Problem: Input/Output Stream & Pipelining



Which input belongs to which output?

Problem: What does ***time*** mean in hardware?



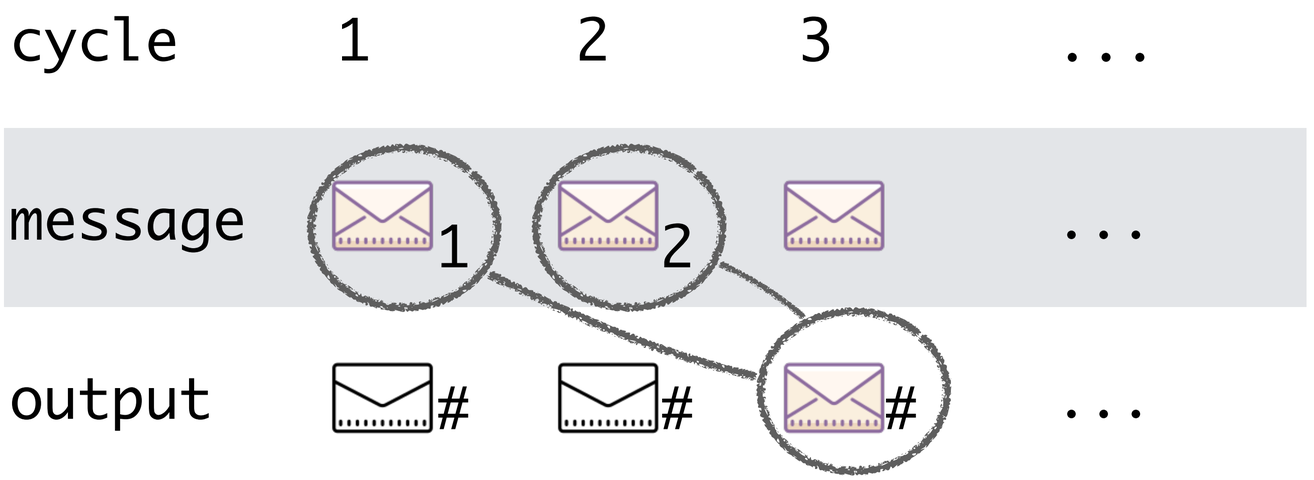
?

Input/Output Stream

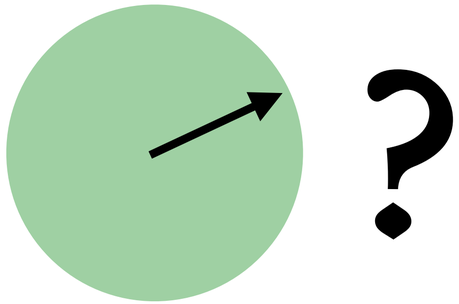
Inputs over Multiple Cycles

Outputs over Multiple Cycles

Problem: *Inputs* over Multiple Cycles



Computing Hash: Multiple Inputs



Input/Output Stream

Inputs over Multiple Cycles

Outputs over Multiple Cycles

Problem: *Outputs* over Multiple Cycles

cycle	1	...
number	10	...
output	0	...

Countdown: Multiple Outputs

Problem: *Outputs* over Multiple Cycles

cycle	1	2	...
number	10	0	...
output	0	10	...

Countdown: Multiple Outputs

Problem: *Outputs* over Multiple Cycles

cycle	1	2	3	...
number	10	0	0	...
output	0	10	9	...

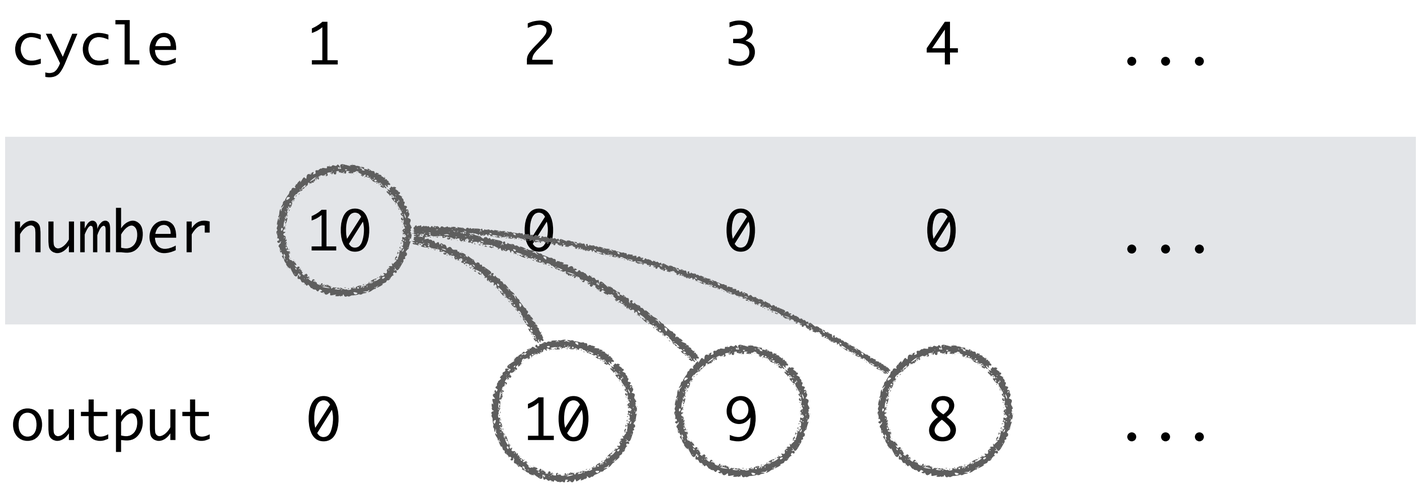
Countdown: Multiple Outputs

Problem: *Outputs* over Multiple Cycles

cycle	1	2	3	4	...
number	10	0	0	0	...
output	0	10	9	8	...

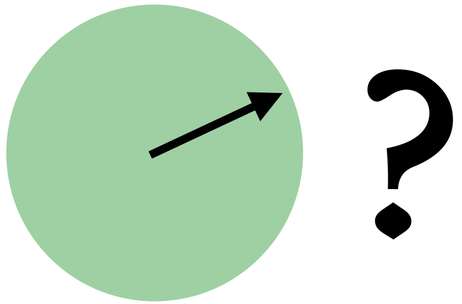
Countdown: Multiple Outputs

Problem: *Outputs* over Multiple Cycles



Countdown: Multiple Outputs

Which input belongs to which output?



?

Input/Output Stream

Inputs over Multiple Cycles

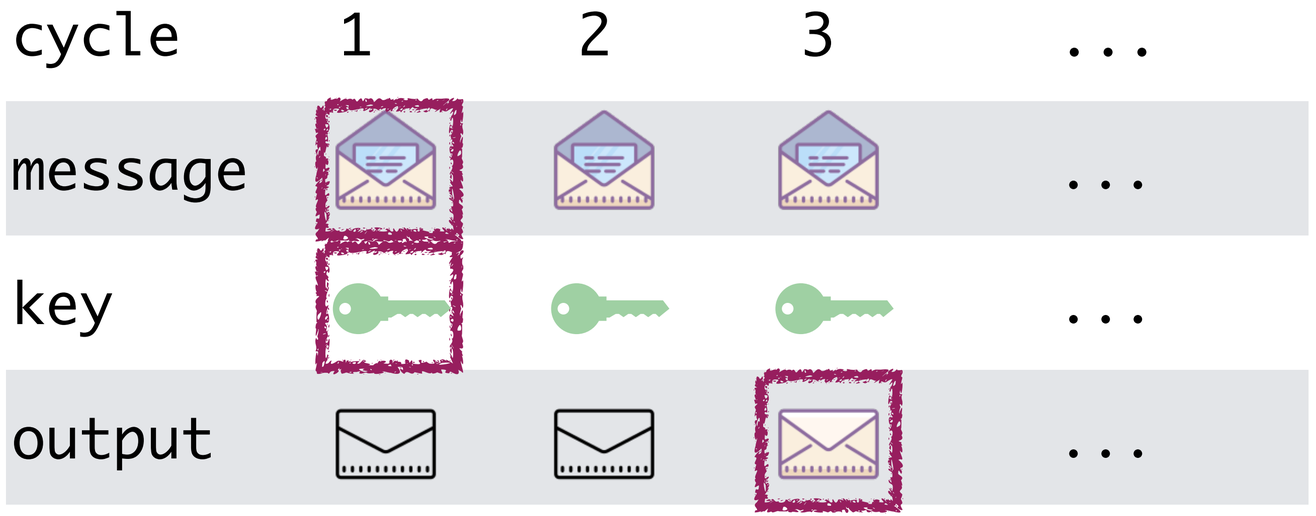
Outputs over Multiple Cycles

Idea: IODINE

Assign a **color** to all inputs at the same ***cycle***

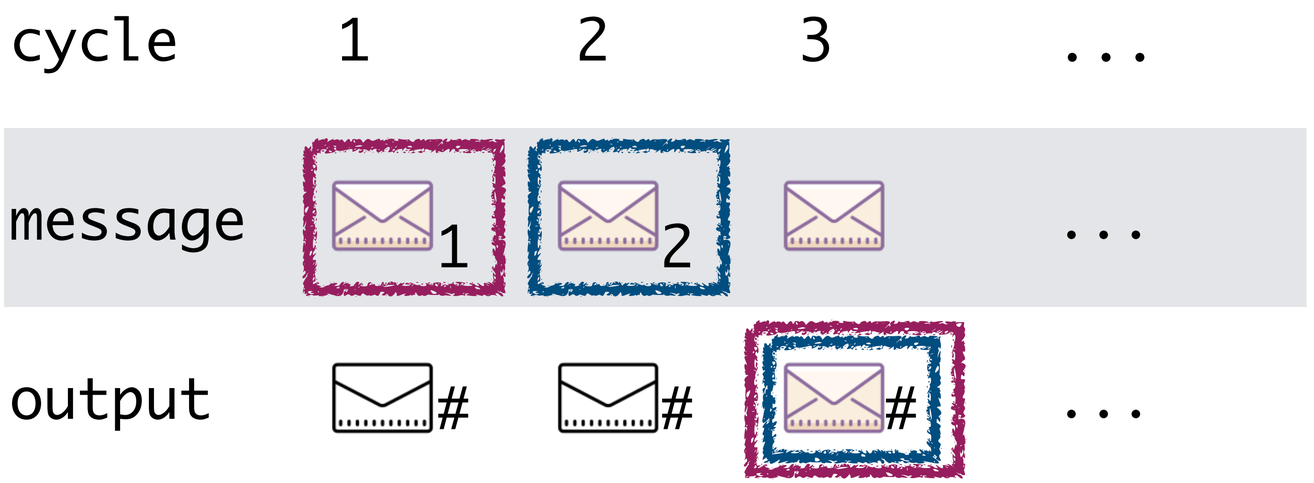
… and track the flow of **colors** in the circuit

Assign a **color** to all inputs at the same **cycle**



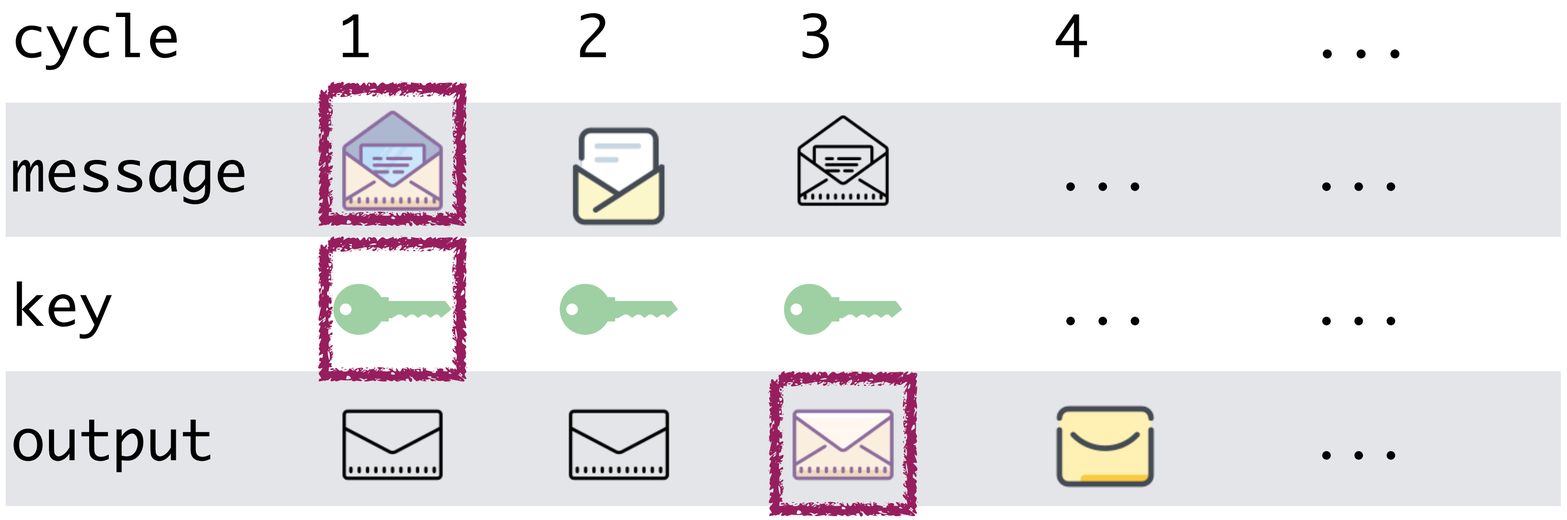
… and track the flow of **colors** in the circuit

Assign a **color** to all inputs at the same **cycle**



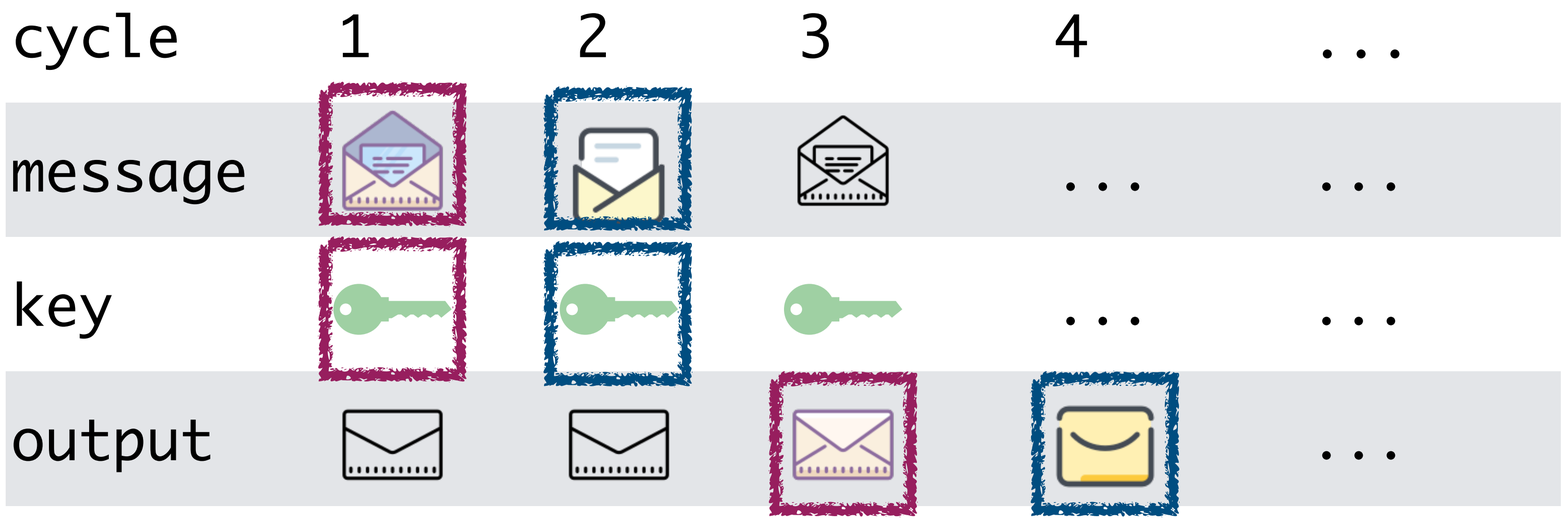
… and track the flow of **colors** in the circuit

Assign a **color** to all inputs at the same **cycle**



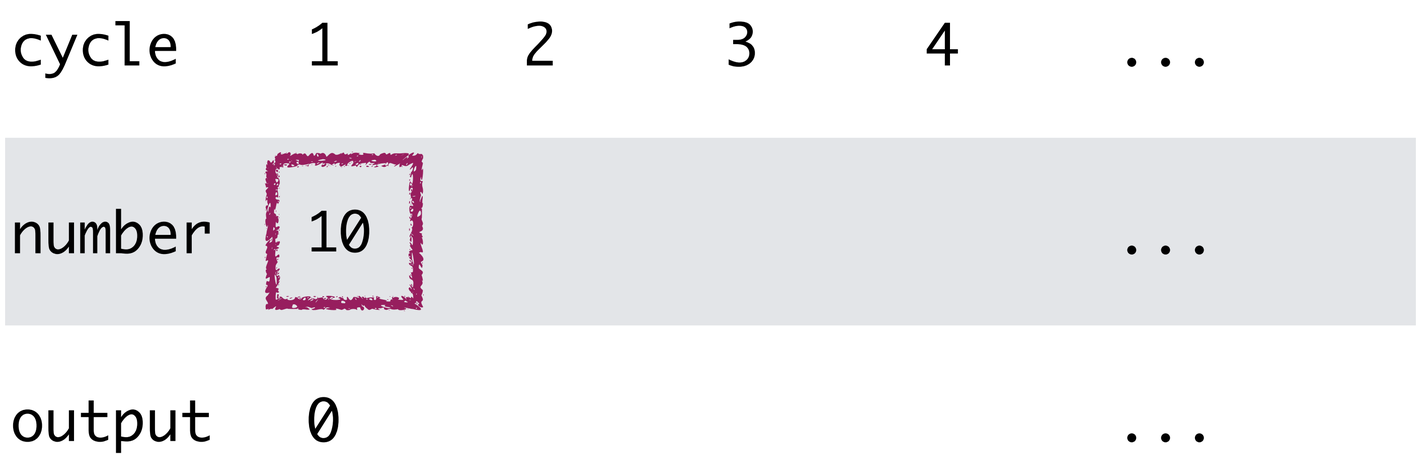
… and track the flow of **colors** in the circuit

Assign a **color** to all inputs at the same **cycle**



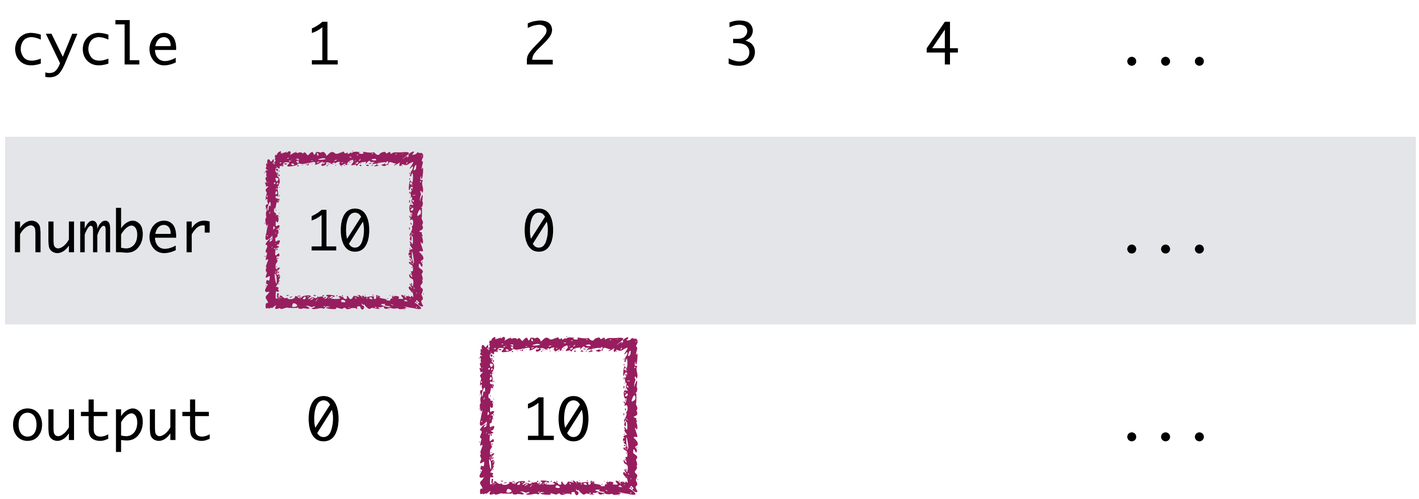
… and track the flow of **colors** in the circuit

Assign **color** to inputs at the same **cycle**



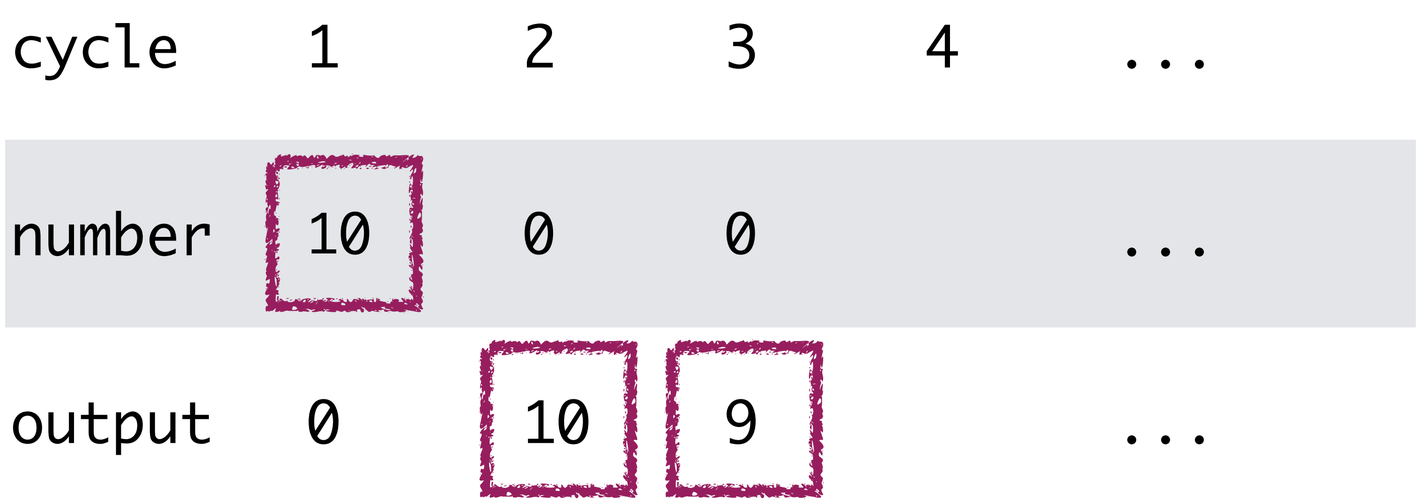
… and track the flow of **colors** in the circuit

Assign **color** to inputs at the same **cycle**



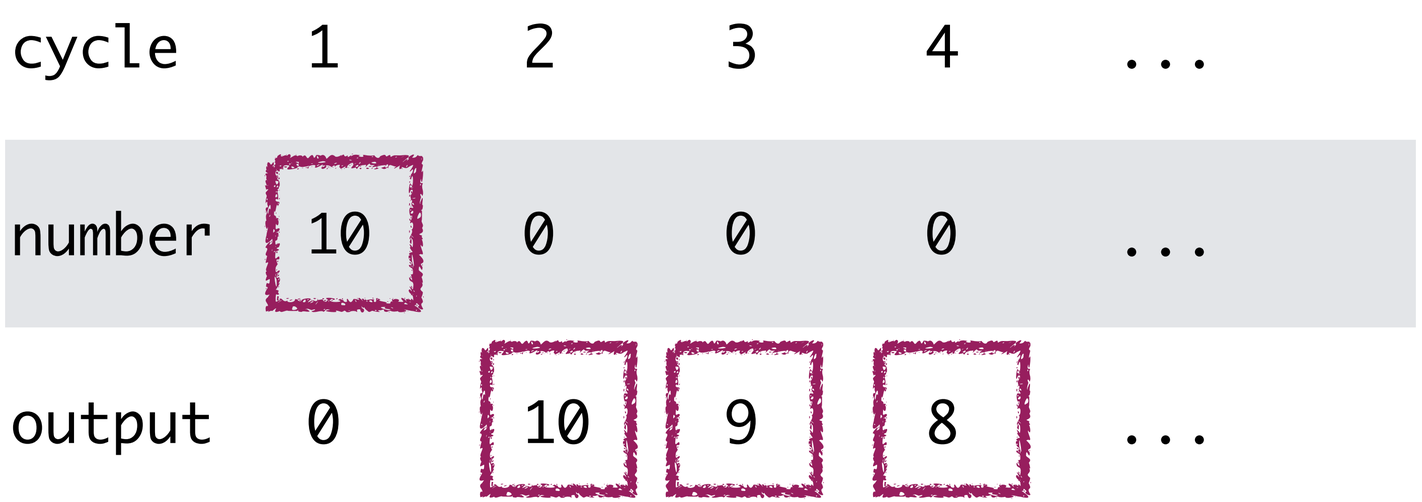
… and track the flow of **colors** in the circuit

Assign **color** to inputs at the same **cycle**



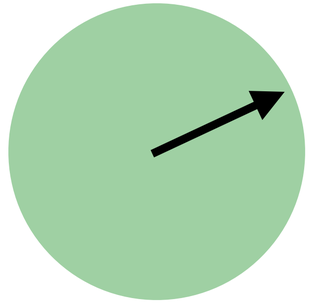
… and track the flow of **colors** in the circuit

Assign **color** to inputs at the same **cycle**



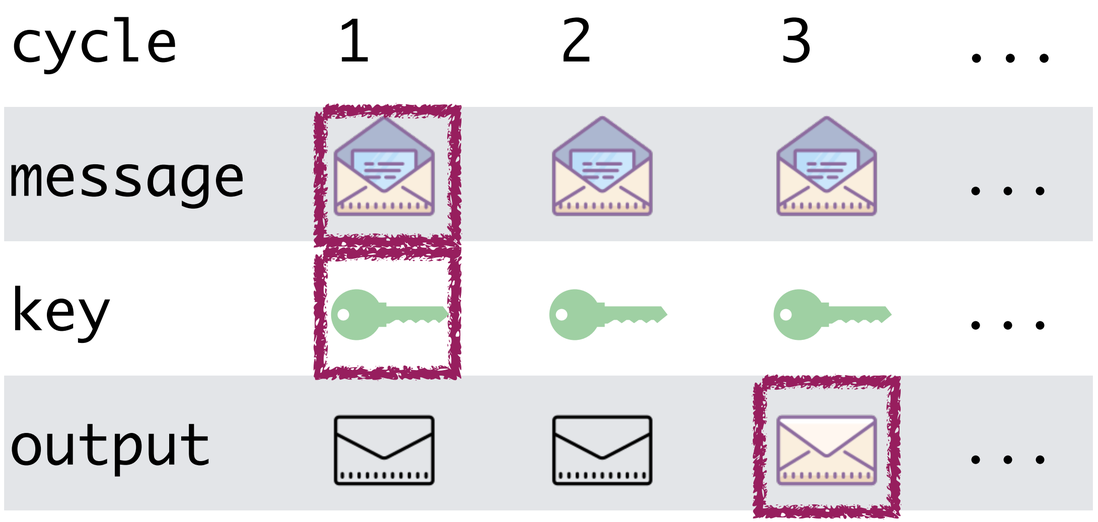
… and track the flow of **colors** in the circuit

Constant Time Definition via *Coloring*

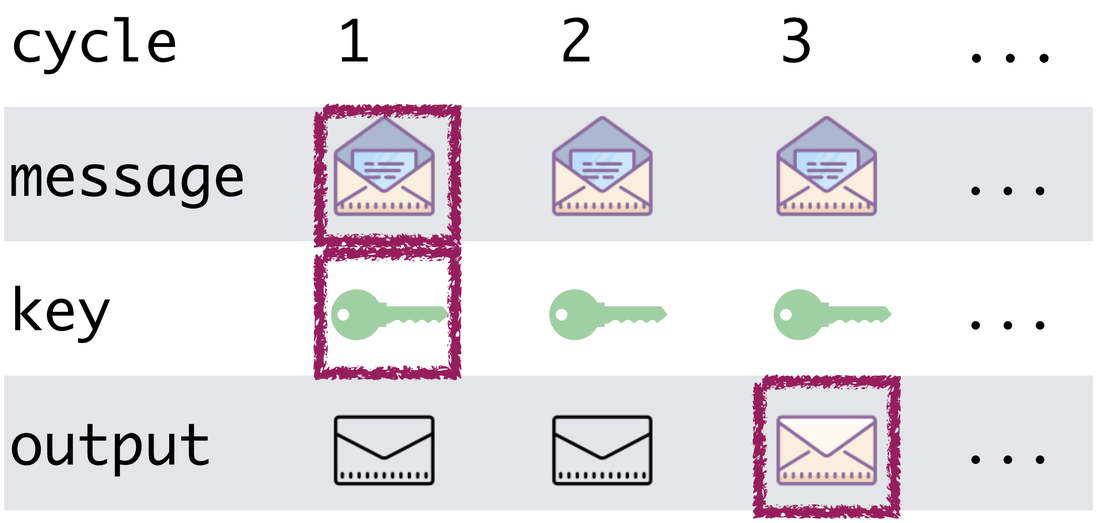


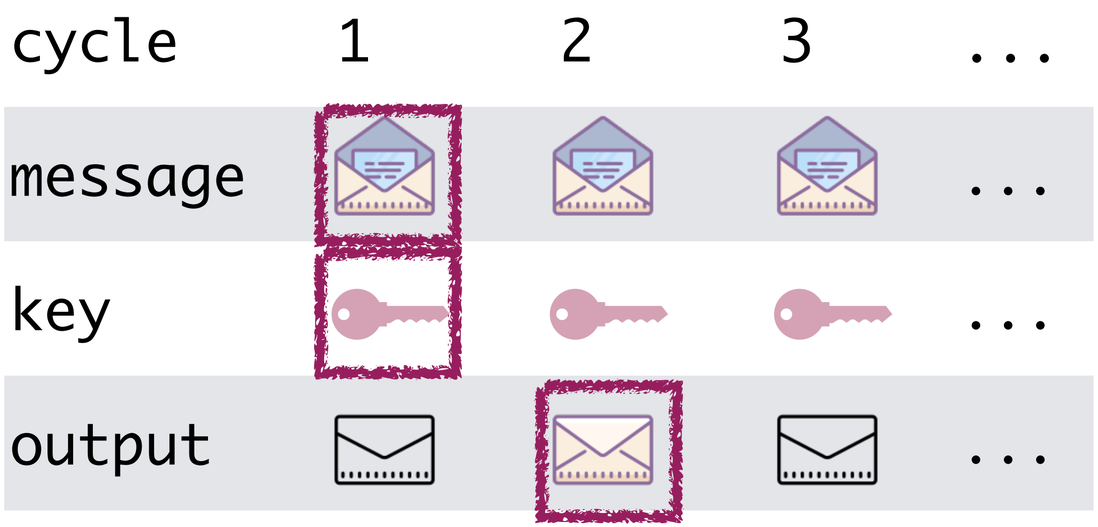
Constant time:
if for ***any*** two executions,
outputs have the **same colors**

Catching a Timing Attack via Coloring



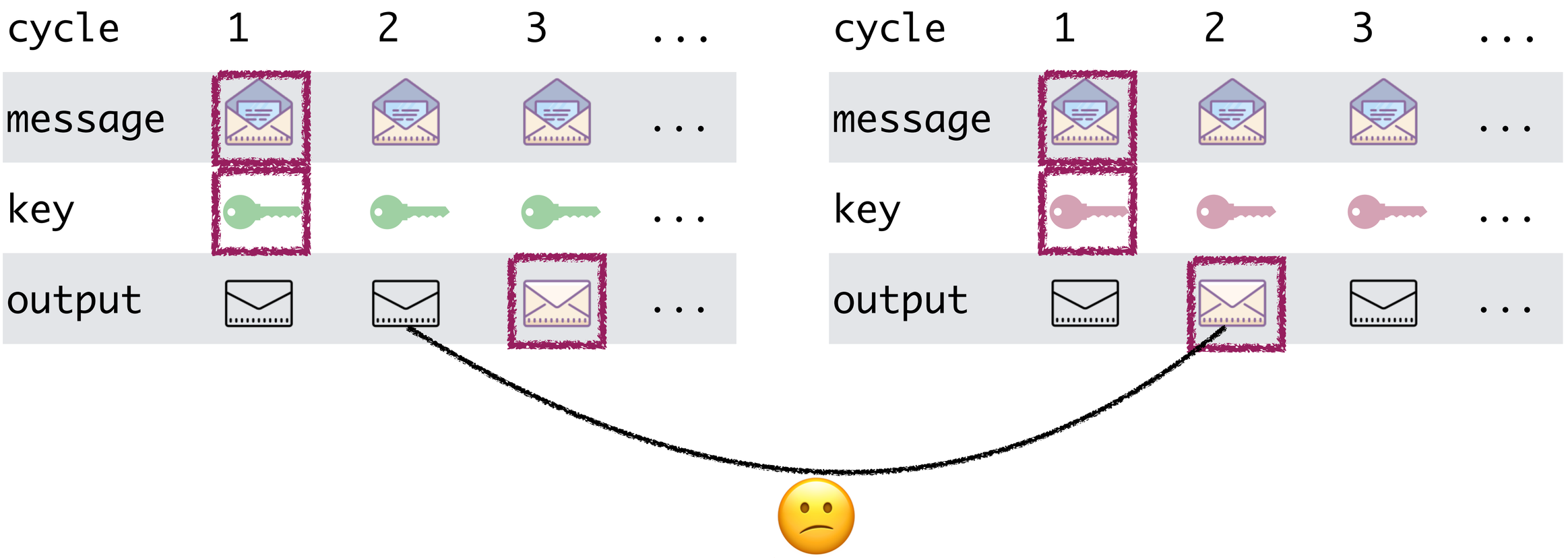
Catching a Timing Attack via Coloring





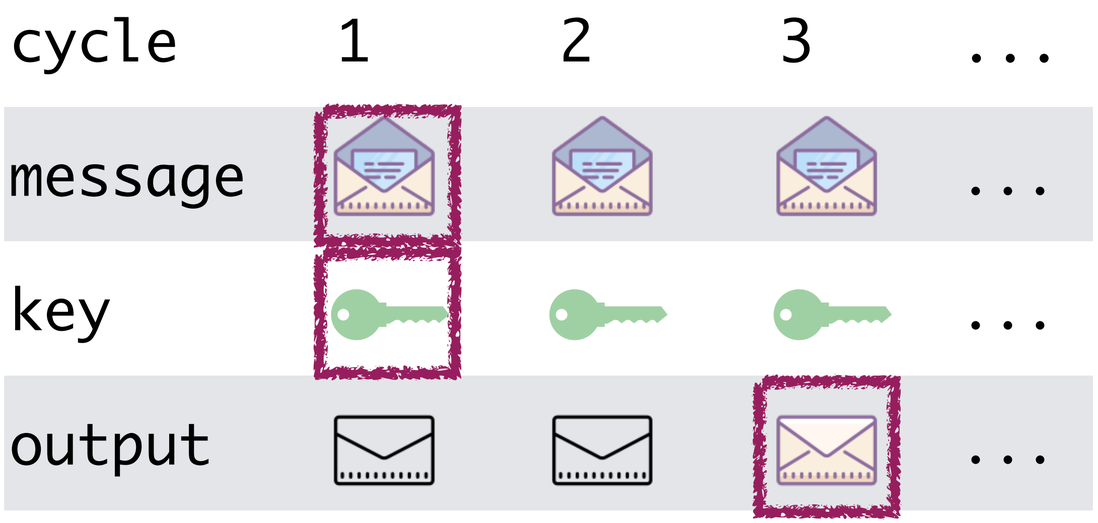
Catching a Timing Attack via Coloring

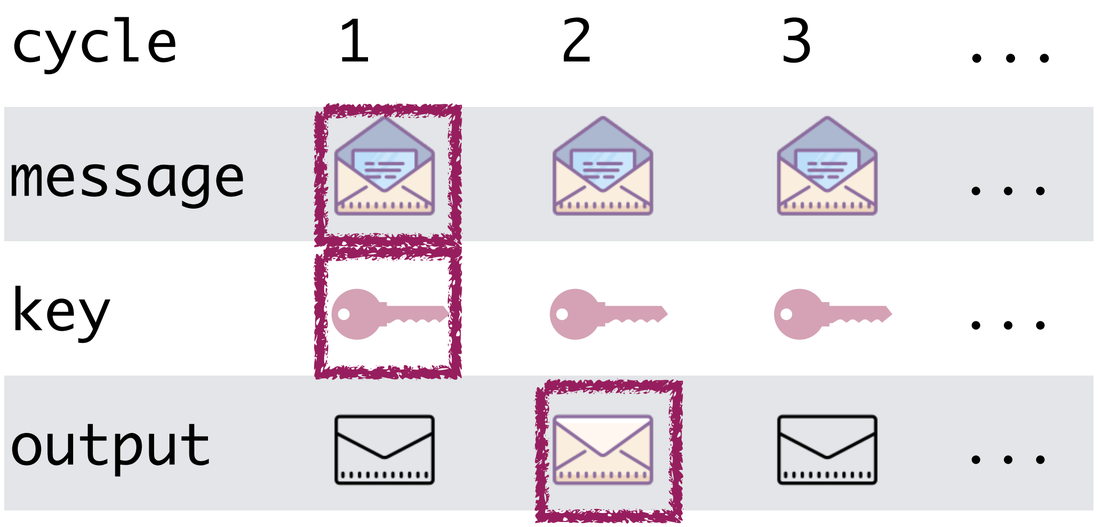
Not constant time!



Catching a Timing Attack via Coloring

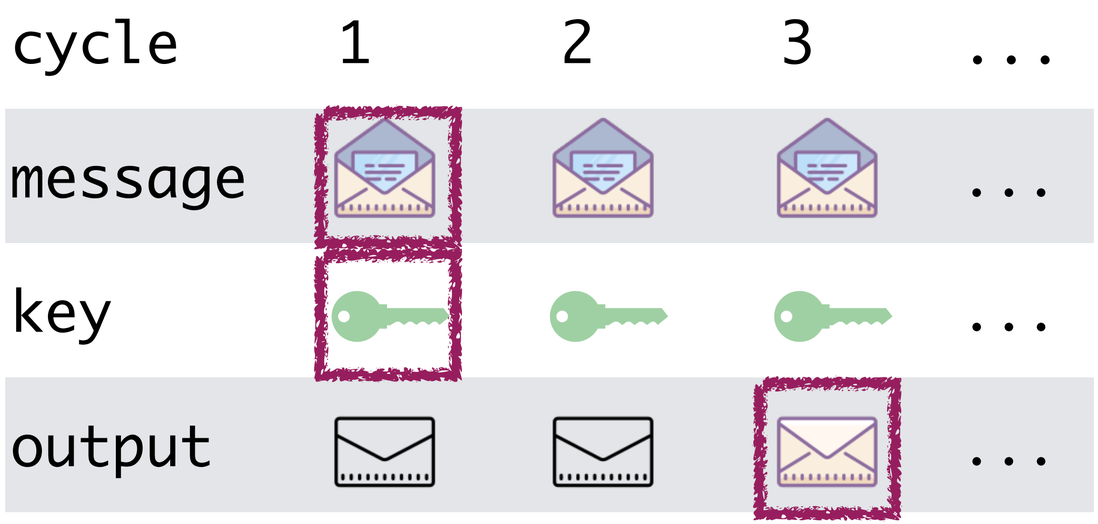
Not constant time!

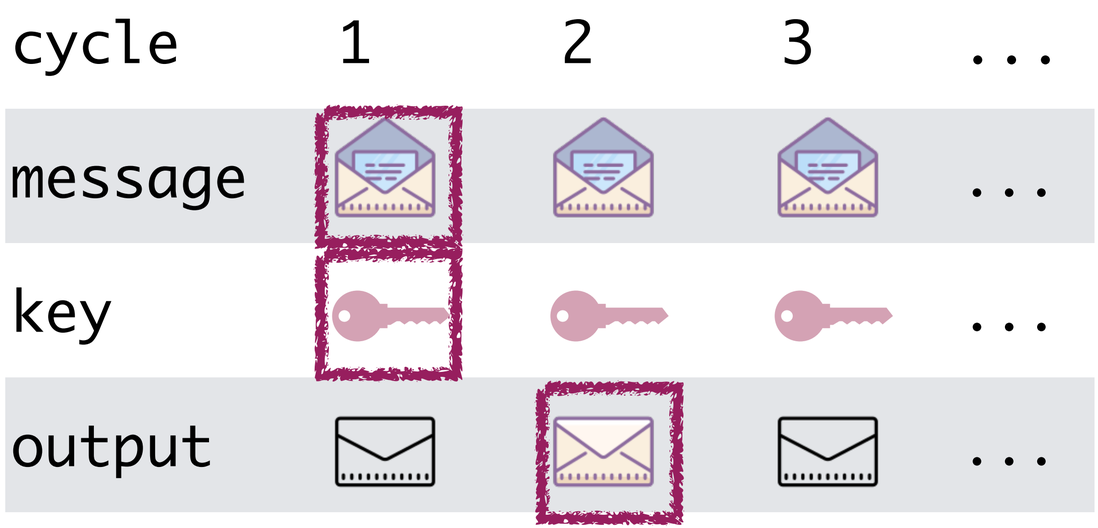




… fixing the circuit, yields

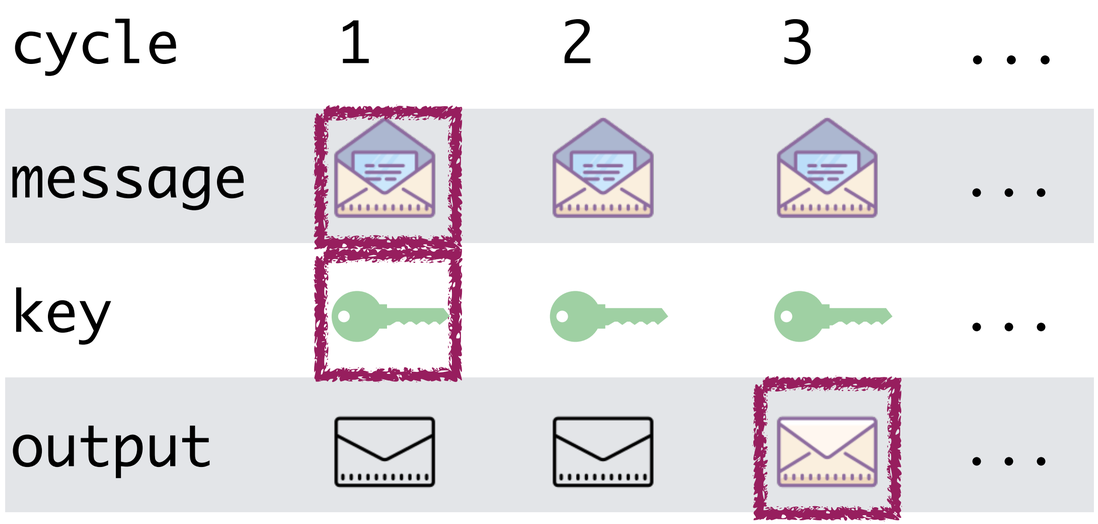
Catching a Timing Attack via Coloring

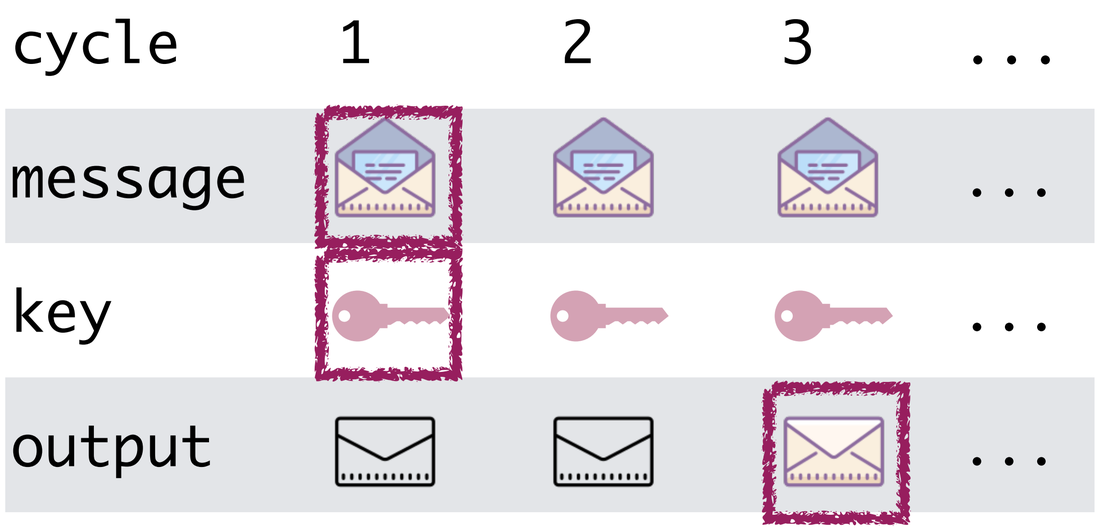




… fixing the circuit, yields

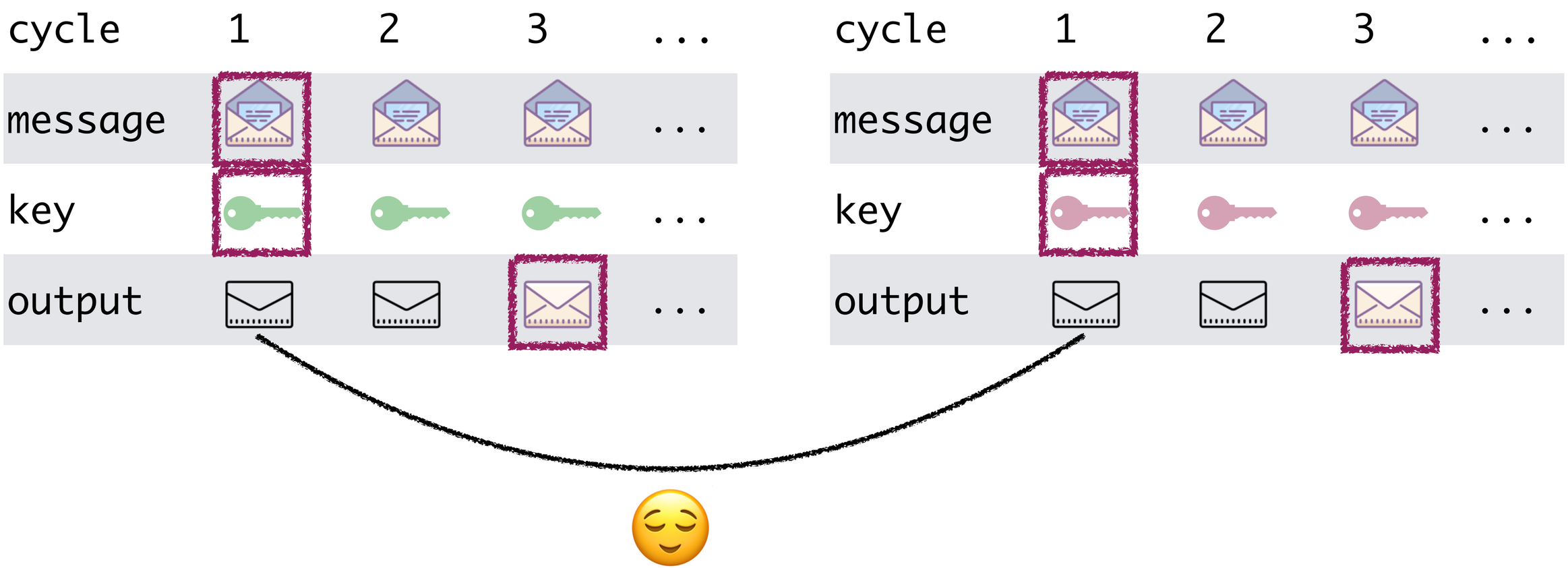
Constant Time via Coloring



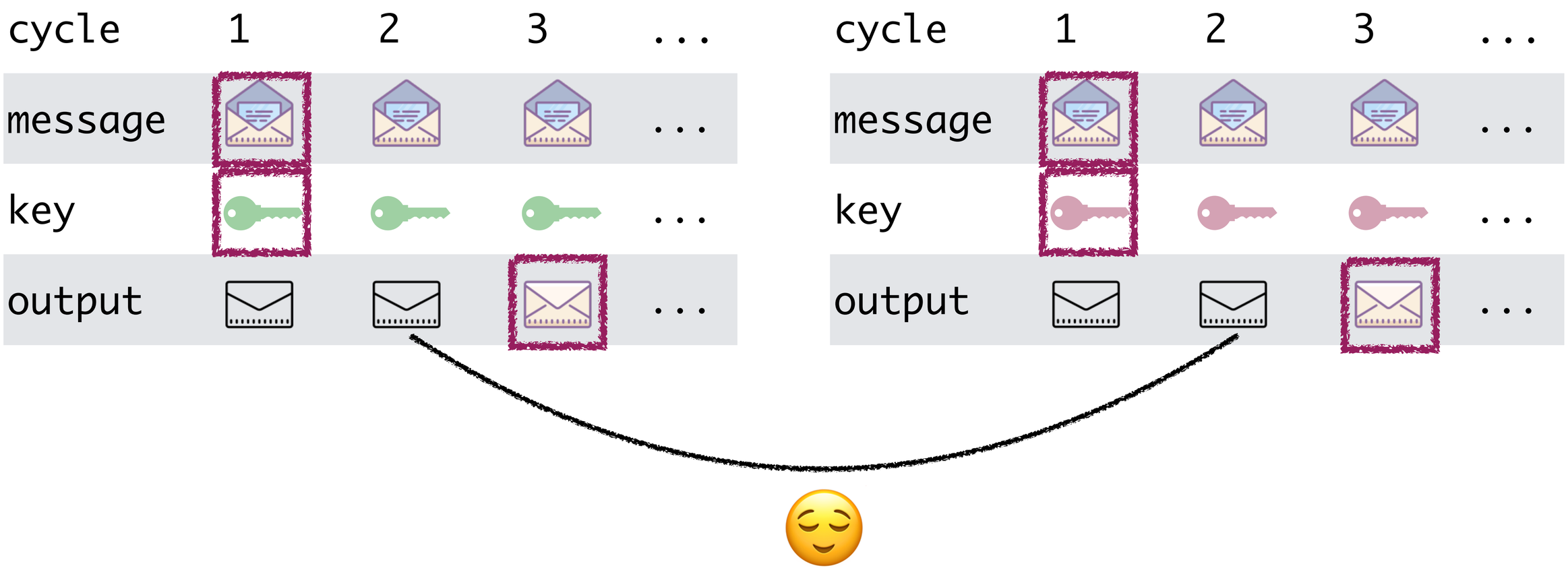


... fixing the circuit, yields

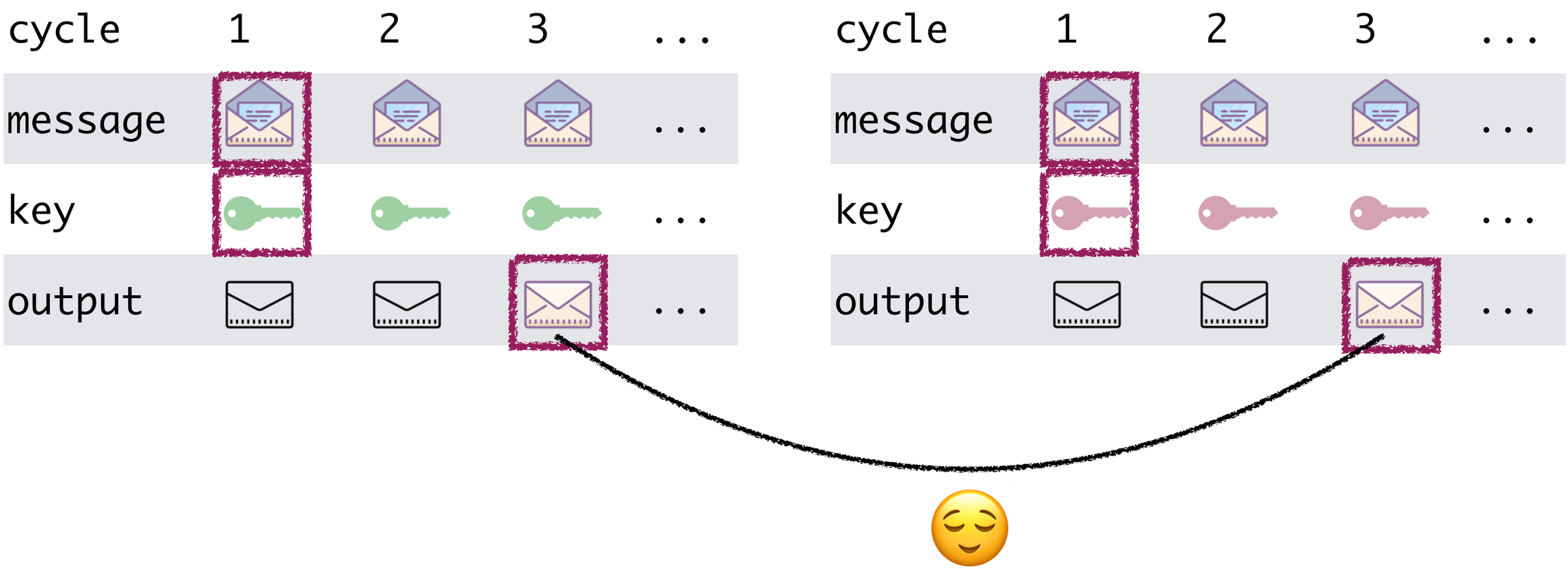
Constant Time via Coloring



Constant Time via Coloring



Constant Time via Coloring



IODINE: What's timing in Hardware?

Specifying **Constant time** for Hardware

Verifying **Constant time** for Hardware

Applying **IODINE**

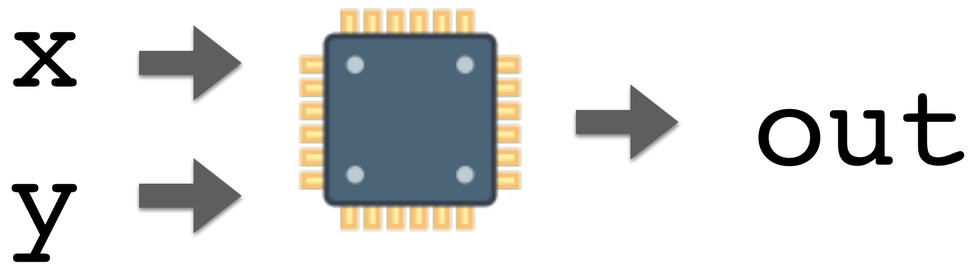
IODINE: What's timing in Hardware?

Specifying **Constant time** for Hardware

Verifying **Constant time** for Hardware

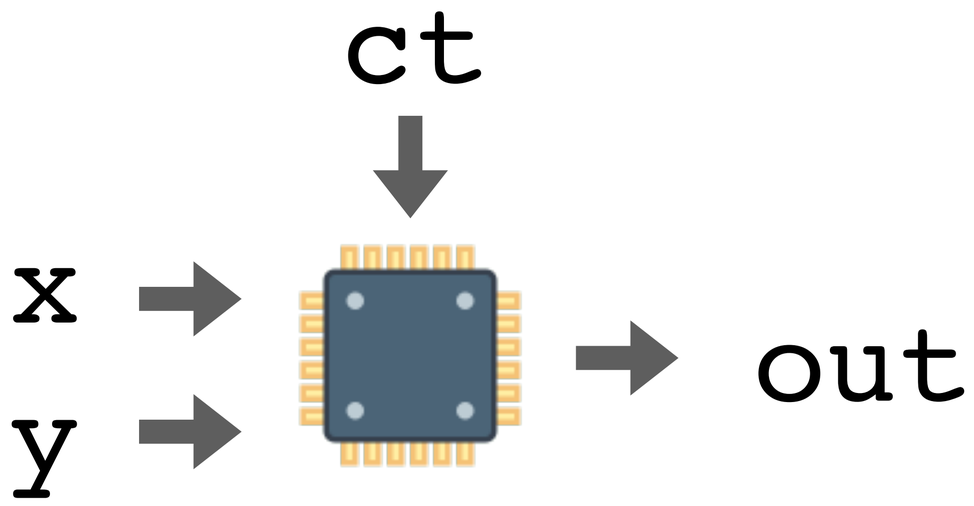
Applying IODINE

Example: MUL x y



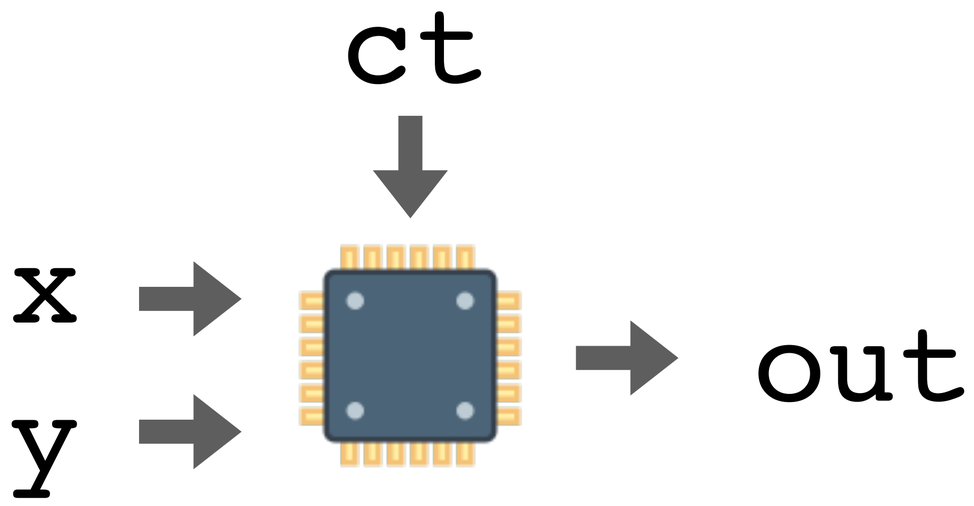
fast-path, if x or y is zero

Example: MUL x y



if ct is set, **don't** take fast path

Example: MUL x y



Implemented in **ARM DIT**

Example: MUL x y

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero && !ct)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

Always Blocks

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero && !ct)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

Run in parallel

Always Blocks

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero && !ct)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

Run in parallel, shared clock

Always Blocks

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero && !ct)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

Write to different variables

Nonblocking Assignment

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero && !ct)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

LHS gets update in next cycle

Continuous Assignment

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero && !ct)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

LHS updated as soon as RHS changes

Take fast path, if an input is zero and not ct

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero && !ct)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

Take fast path, if an input is zero and not ct

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero && !ct)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

else, assign result from slow path

Compute slow path result

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero && !ct)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

Example: MUL x y with **ct=0**

cycle	1	2	···	k
x	1			
y	1			
out	⊥			

Slow path

Example: MUL x y with **ct=0**

cycle	1	2	···	k
x	1	1		
y	1	1		
out	⊥	⊥		

Slow path

Example: MUL x y with **ct=0**

cycle	1	2	···	k
x	1	1	···	1
y	1	1	···	1
out	⊥	⊥	···	1

Slow path

Example: MUL x y with **ct=0**

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

Slow path

cycle	1	2	⋯	k
x	0			
y	1			
out	⊥			

Fast path

Example: MUL x y with **ct=0**

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

Slow path

cycle	1	2	⋯	k
x	0	0		
y	1	1		
out	⊥	0		

Fast path

Example: MUL x y with **ct=0**

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

Slow path

cycle	1	2	⋯	k
x	0	0	⋯	0
y	1	1	⋯	1
out	⊥	0	⋯	0

Fast path

Example: MUL x y with **ct=0**

Not constant time!

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

cycle	1	2	⋯	k
x	0	0	⋯	0
y	1	1	⋯	1
out	⊥	0	⋯	0

Example: MUL x y with **ct=1**

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

Slow path

cycle	1	2	⋯	k
x	0			
y	1			
out	⊥			

Slow path

Example: MUL x y with **ct=1**

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

Slow path

cycle	1	2	⋯	k
x	0	0		
y	1	1		
out	⊥	⊥		

Slow path

Example: MUL x y with **ct=1**

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

Slow path

cycle	1	2	⋯	k
x	0	0	0	0
y	1	1	1	1
out	⊥	⊥	⊥	0

Slow path

Example: MUL x y with **ct=1**

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

cycle	1	2	⋯	k
x	0	0	0	0
y	1	1	1	1
out	⊥	⊥	⊥	0

Example: MUL x y with **ct=1**

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

cycle	1	2	⋯	k
x	0	0	0	0
y	1	1	1	1
out	⊥	⊥	⊥	0

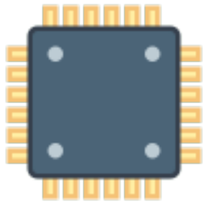
Example: MUL x y with **ct=1**

Is constant time!

cycle	1	2	⋯	k
x	1	1	⋯	1
y	1	1	⋯	1
out	⊥	⊥	⋯	1

cycle	1	2	⋯	k
x	0	0	0	0
y	1	1	1	1
out	⊥	⊥	⊥	0

Verifying Constant Time for Hardware



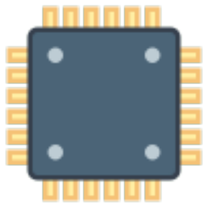
P

P is constant time, if ct=1

Verifying Constant Time for Hardware

cyc	1	2	...	k
x	1	1	...	1
y	1	1	...	1
out	⊥	⊥	..	1

cyc	1	2	...	k
x	0	0	...	0
y	1	1	...	1
out	⊥	⊥	...	0

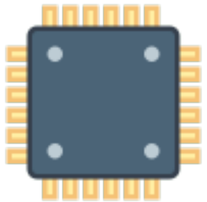


P

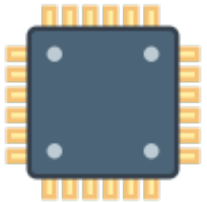
Verifying Constant Time for Hardware

cyc	1	2	...	k
x	1	1	...	1
y	1	1	...	1
out	⊥	⊥	..	1

cyc	1	2	...	k
x	0	0	...	0
y	1	1	...	1
out	⊥	⊥	...	0



P_L

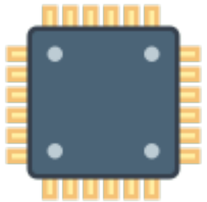


P_R

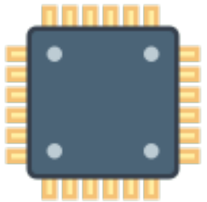
Verifying Constant Time for Hardware

cyc	1	2	...	k
x	1	1	...	1
y	1	1	...	1
out	⊥	⊥	..	1

cyc	1	2	...	k
x	0	0	...	0
y	1	1	...	1
out	⊥	⊥	...	0



P_L

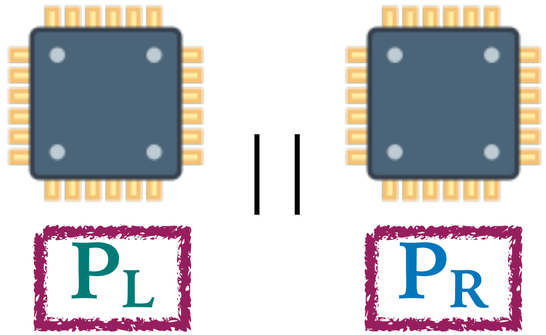


P_R

Verifying Constant Time for Hardware

cyc	1	2	...	k
x	1	1	...	1
y	1	1	...	1
out	⊥	⊥	..	1

cyc	1	2	...	k
x	0	0	...	0
y	1	1	...	1
out	⊥	⊥	...	0



Verifying Constant Time for Hardware

cyc	1	2	...	k
x	1	1	...	1
y	1	1	...	1
out	⊥	⊥	..	1

cyc	1	2	...	k
x	0	0	...	0
y	1	1	...	1
out	⊥	⊥	...	0

$$P_L \;||\; P_R$$

$$\textbf{assert}\left(P_L.\text{out.cols} = P_R.\text{out.cols}\right)$$

Verifying Constant Time for Hardware

cyc	1	2	...	k
x	1	1	...	1
y	1	1	...	1
out	⊥	⊥	..	1

cyc	1	2	...	k
x	0	0	...	0
y	1	1	...	1
out	⊥	⊥	...	0

$$\textbf{assume}\left(P_L.ct = 1 \wedge P_R.ct = 1\right)$$

$$P_L \;||\; P_R$$

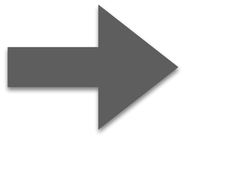
$$\textbf{assert}\left(P_L.\text{out.cols} = P_R.\text{out.cols}\right)$$

Verifying Constant Time for Hardware

$$\textbf{assume}\left(P_L.ct = 1 \wedge P_R.ct = 1\right)$$

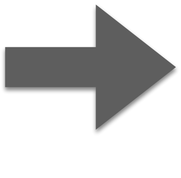
$$P_L \;||\; P_R$$

$$\textbf{assert}\left(P_L.out.cols = P_R.out.cols\right)$$

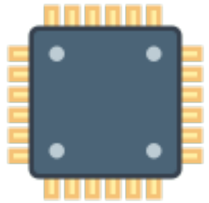


$\exists$:  ...

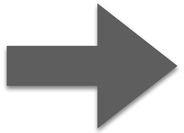
Horn Clauses



Verifying Constant Time for Hardware



P



init(P)

next(P, P’)

Verifying Constant Time for Hardware

$$\mathrm{init}(P) \quad = \quad \mathrm{true}$$

Verifying Constant Time for Hardware

$$\text{init}(P) \quad = \quad \text{true}$$

$$\text{next}(P, P') \quad = \quad \left(\text{ct} = 0 \wedge \text{iszero} = 1 \rightarrow \text{out}' = 0 \right)$$

Verifying Constant Time for Hardware

$$\mathrm{init}(P) \quad = \quad \mathrm{true}$$

$$\mathrm{next}(P, P') \quad = \quad \begin{pmatrix} \mathrm{ct}=0 \wedge \mathrm{iszero}=1 \rightarrow \mathrm{out}'=0 \\ \wedge \ \ldots \rightarrow \mathrm{out}'=\mathrm{flp\_res} \end{pmatrix}$$

Verifying Constant Time for Hardware

$$\text{init}(P) \quad = \quad \text{true}$$

$$\text{next}(P, P') \quad = \quad \begin{pmatrix} ct=0 \quad iszero=1 \rightarrow out'=0 \\ \wedge \ \ldots \rightarrow out'=flp\_res \end{pmatrix}$$

$$\wedge \quad flp\_res' = \ldots$$

Verifying Constant Time for Hardware

$$\text{init}(P) \quad = \quad \text{true}$$

$$\text{next}(P, P') \quad = \quad \begin{pmatrix} ct=0 \quad iszero=1 \rightarrow out'=0 \\ \wedge \ \ldots \rightarrow out'=flp\_res \end{pmatrix}$$

$$\wedge \quad flp\_res' = \ldots$$

$$\wedge \quad iszero = (x=0 \vee y=0)$$

Verifying Constant Time for Hardware

$$\exists: \text{inv}\,(P_L, P_R)$$

$$P_L.\text{ct} = 1 \;\wedge\; P_L.\text{ct} = 1 \wedge \text{init}(P_L) \wedge \text{init}(P_R) \;\rightarrow\; \text{inv}\,(P_L, P_R)$$

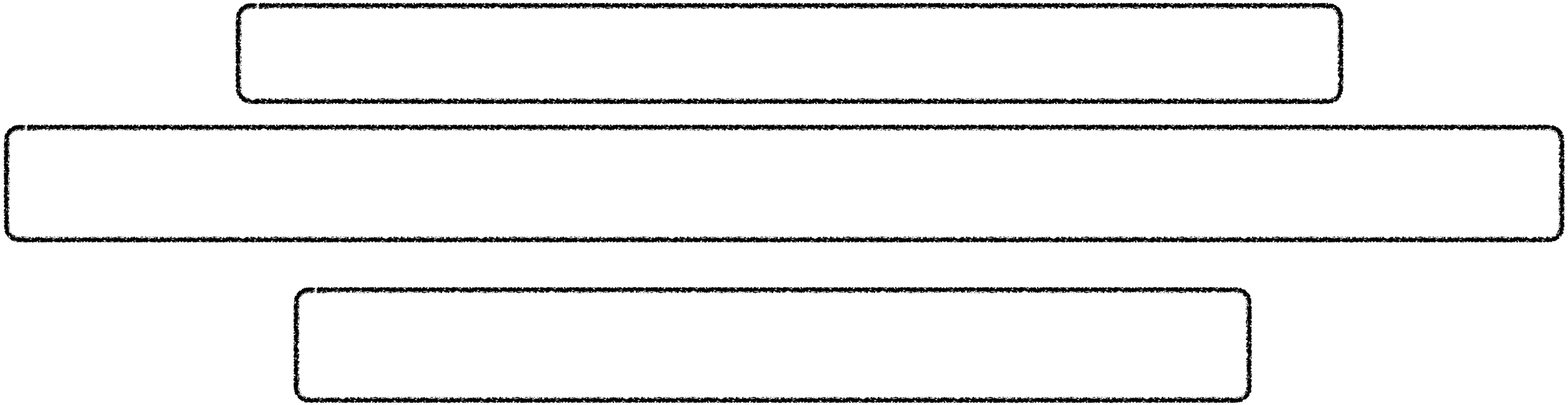
$$\begin{pmatrix} P_L.\text{ct} = 1 \\ P_L.\text{ct} = 1 \end{pmatrix} \wedge \text{inv}\,(P_L, P_R) \wedge \text{next}(P_L, P_L') \wedge \text{next}(P_R, P_R') \rightarrow \text{inv}\,(P_L', P_R')$$

$$\text{inv}\,(P_L, P_R) \rightarrow \left(P_L.\text{out.cols} = P_R.\text{out.cols} \right)$$

Horn Clauses

Verifying Constant Time for Hardware

∃: inv (P_L, P_R)



Horn Clauses

Verifying Constant Time for Hardware

$\exists$: inv (P_L, P_R)

IODINE: What's timing in Hardware?

Specifying **Constant time** for Hardware

Verifying **Constant time** for Hardware

Applying IODINE

IODINE: What's timing in Hardware?

Specifying **Constant time** for Hardware

Verifying **Constant time** for Hardware

Applying **IODINE**

Applying IODINE: Benchmarks

472 MIPS	✓	 Simple CPUs
Yarvi RISC-V	✓	
Single precision FPU	✓	 ALU/FPU
IEEE 754 FPU	✗	
TIS-CT ALU	✓	
Opencores Shacore SHA-256	✓	 Crypto Cores
Fatestudio RSA 4096 RSA	✗	

My Journey

[1] Iodine: What's timing in HW. **Usenix Security**'19.

[2] Xenon: Help non-experts find assumptions. **CCS**'21.

[3] LeaVe: Assumptions at Software level. **CCS**'23.
Distinguished paper.

My Journey

[1] Iodine: What's timing in HW. **Usenix Security**'19.

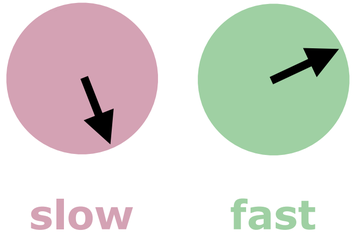
[2] Xenon: Help non-experts find assumptions. **CCS**'21.

[3] LeaVe: Assumptions at Software level. **CCS**'23. **Distinguished paper**.

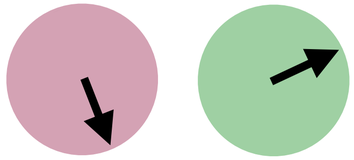
Assumptions in IODINE

Programming Model

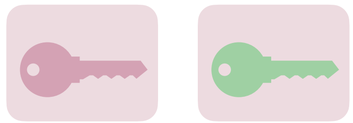
(1) `div` 



(2) `if`  `{…}`

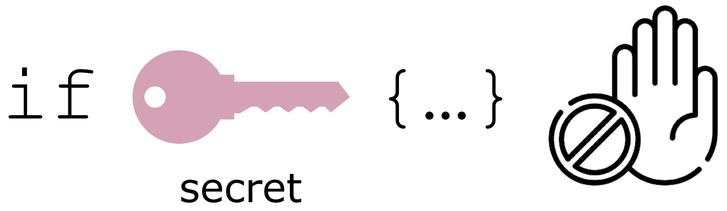


(3) `a[`  `]`



Processors are
only safe
under certain assumptions

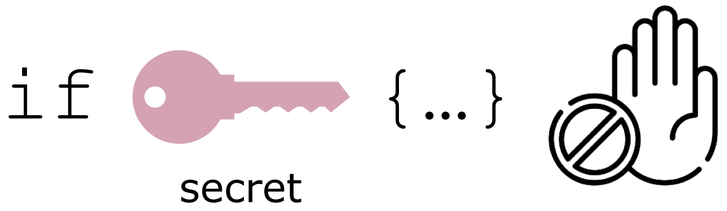
Assumptions in IODINE



$$\textbf{assume}\left(\mathbf{P_L}.\text{instruction} = \mathbf{P_R}.\text{instruction}\right)$$

$$\textbf{assume}\left(\mathbf{P_L}.\text{pc} = \mathbf{P_R}.\text{pc}\right)$$

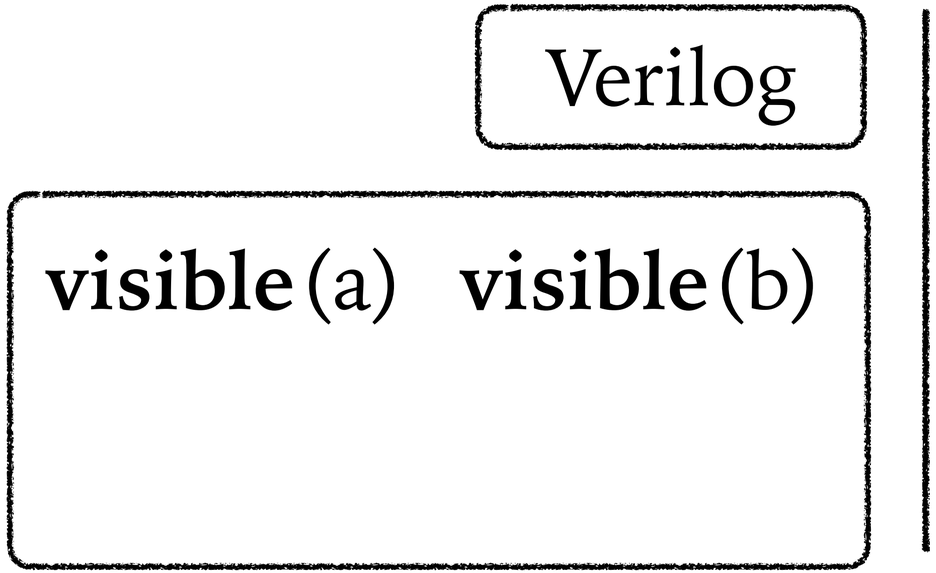
Assumptions in IODINE



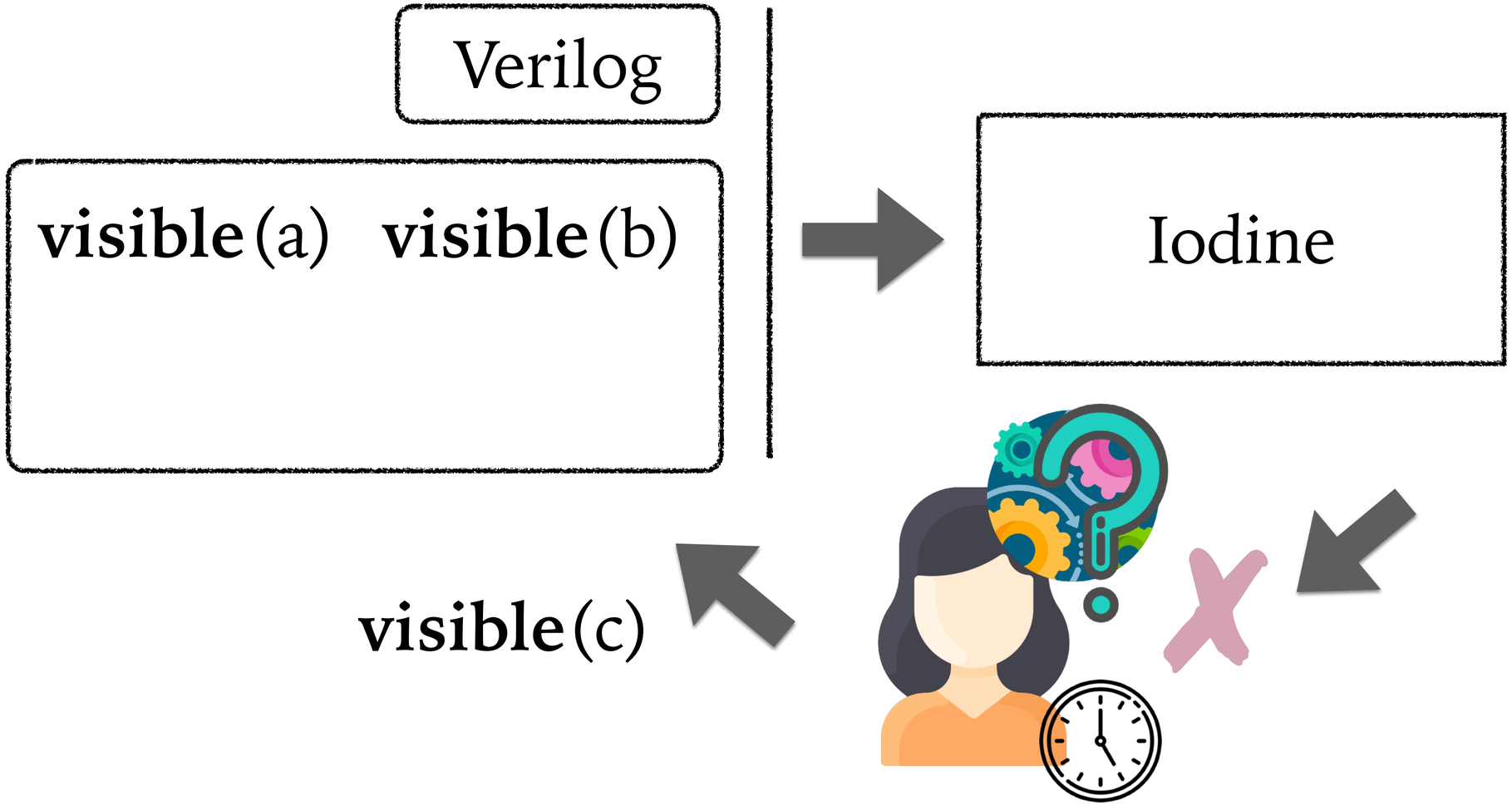
visible (instruction)

visible (pc)

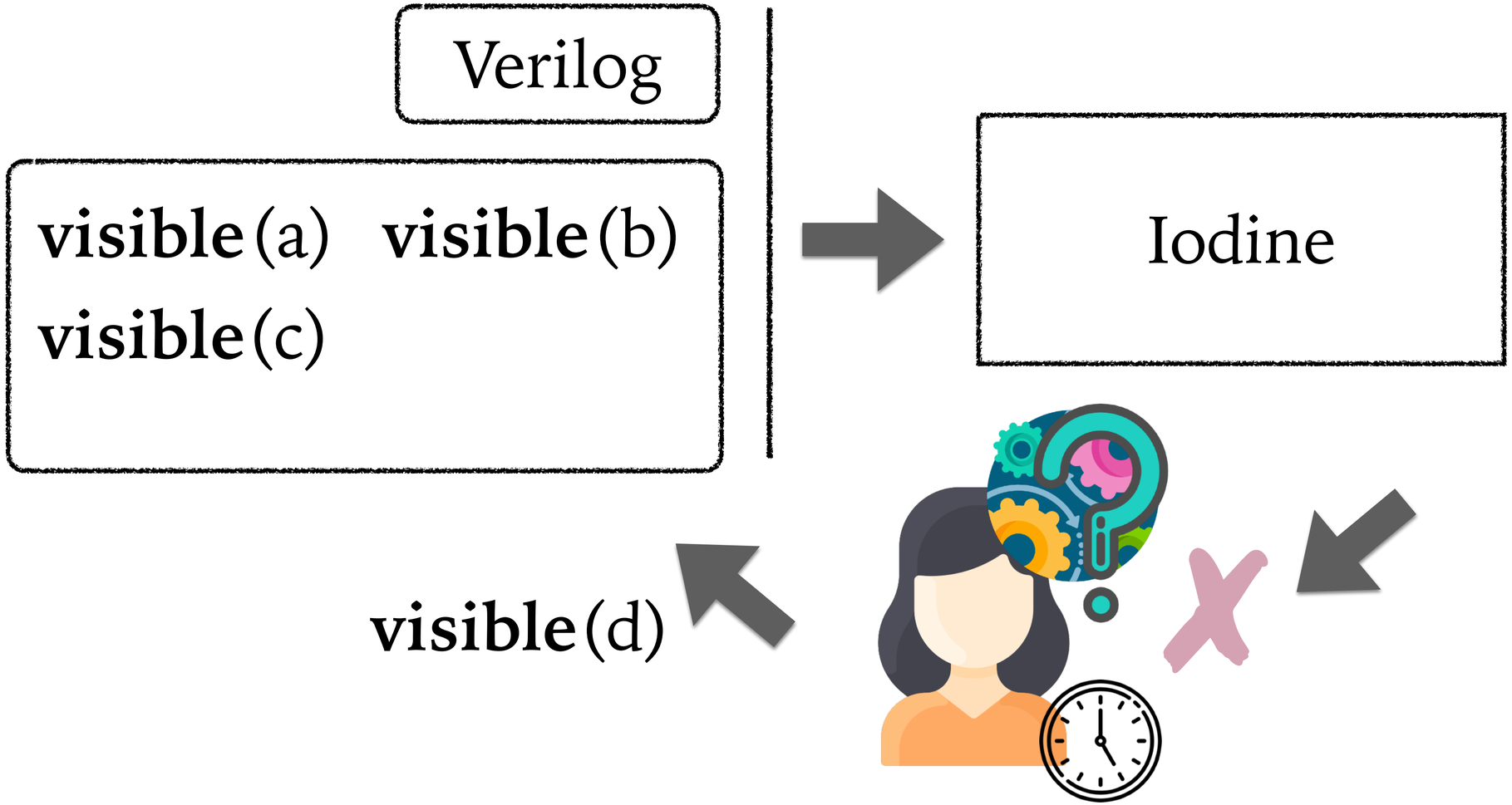
Problem: How to find Assumptions?



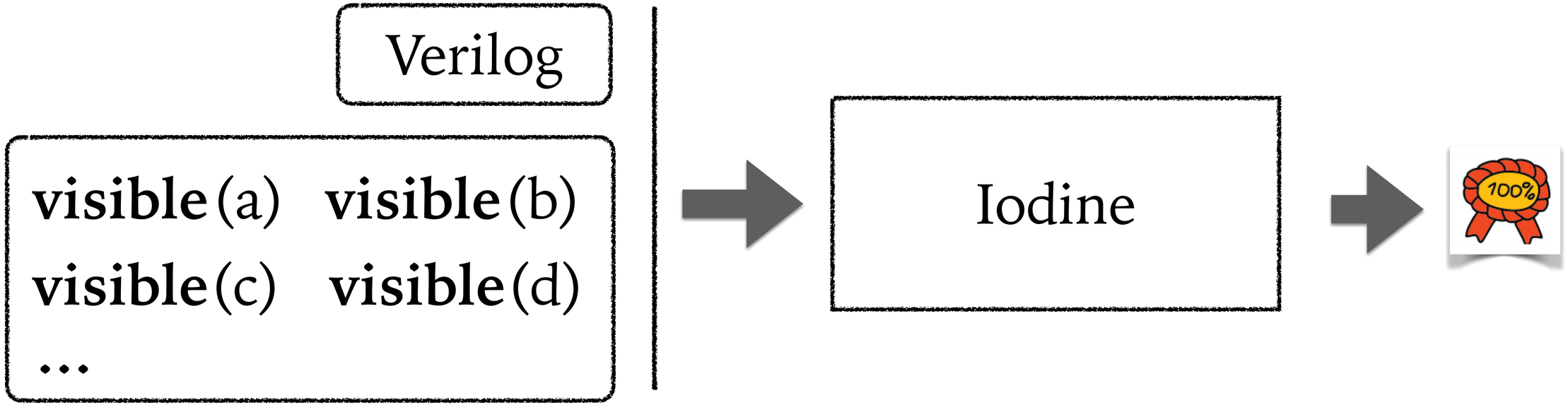
Problem: How to find Assumptions?

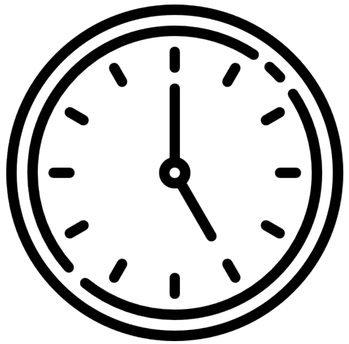


Problem: How to find Assumptions?



Problem: How to find Assumptions?

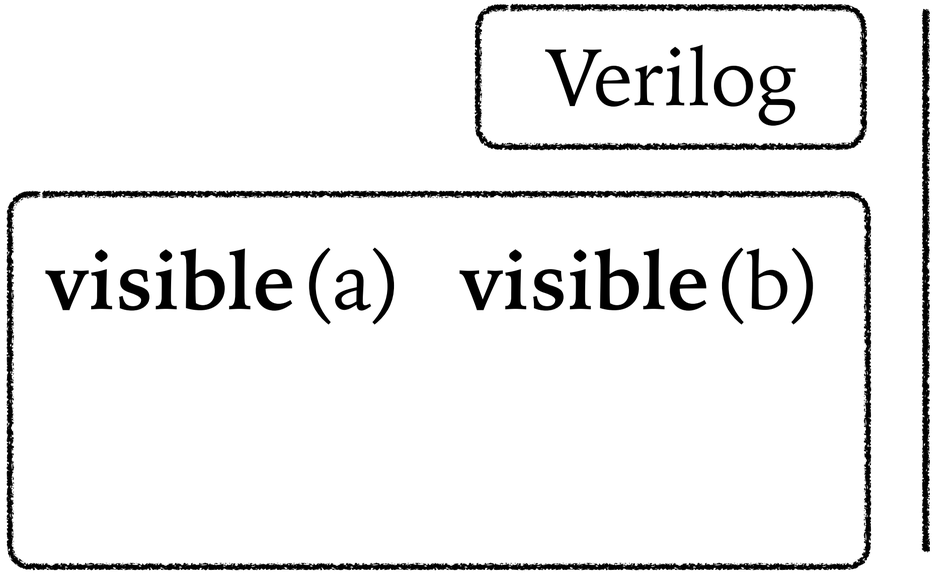




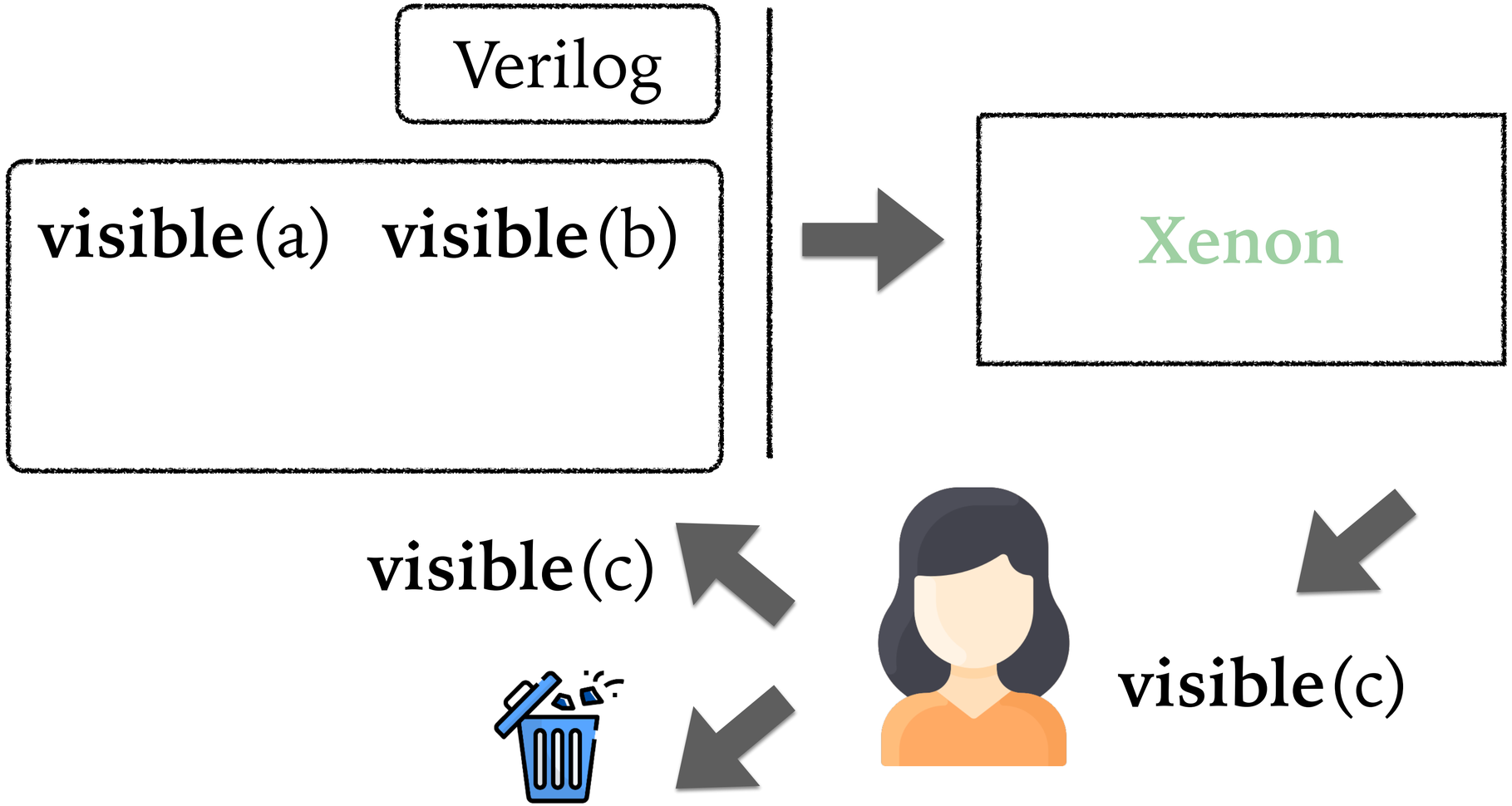
Finding missing assumptions

makes up most of verification effort

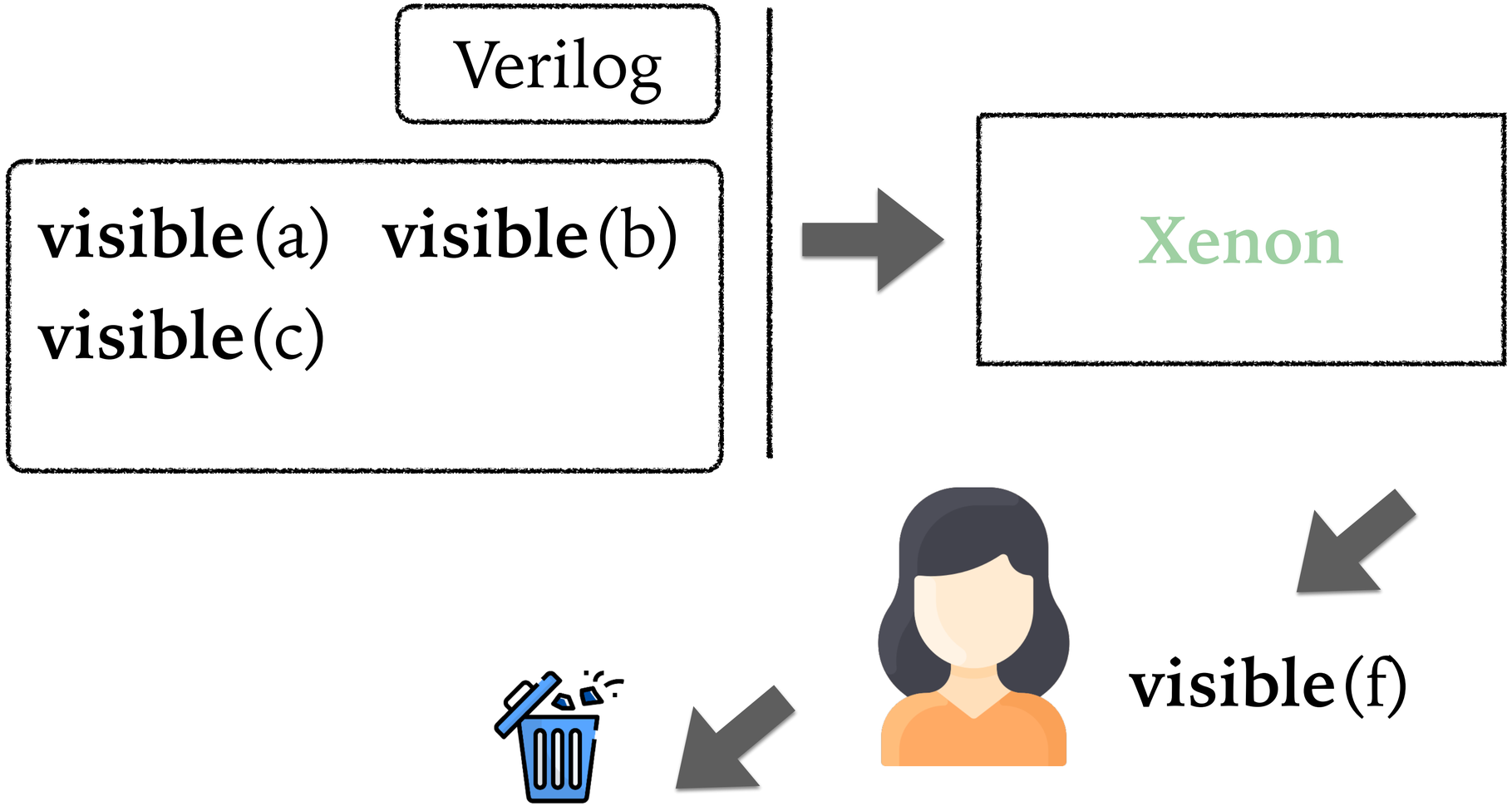
Xenon: Solver-Aided Assumption Synthesis



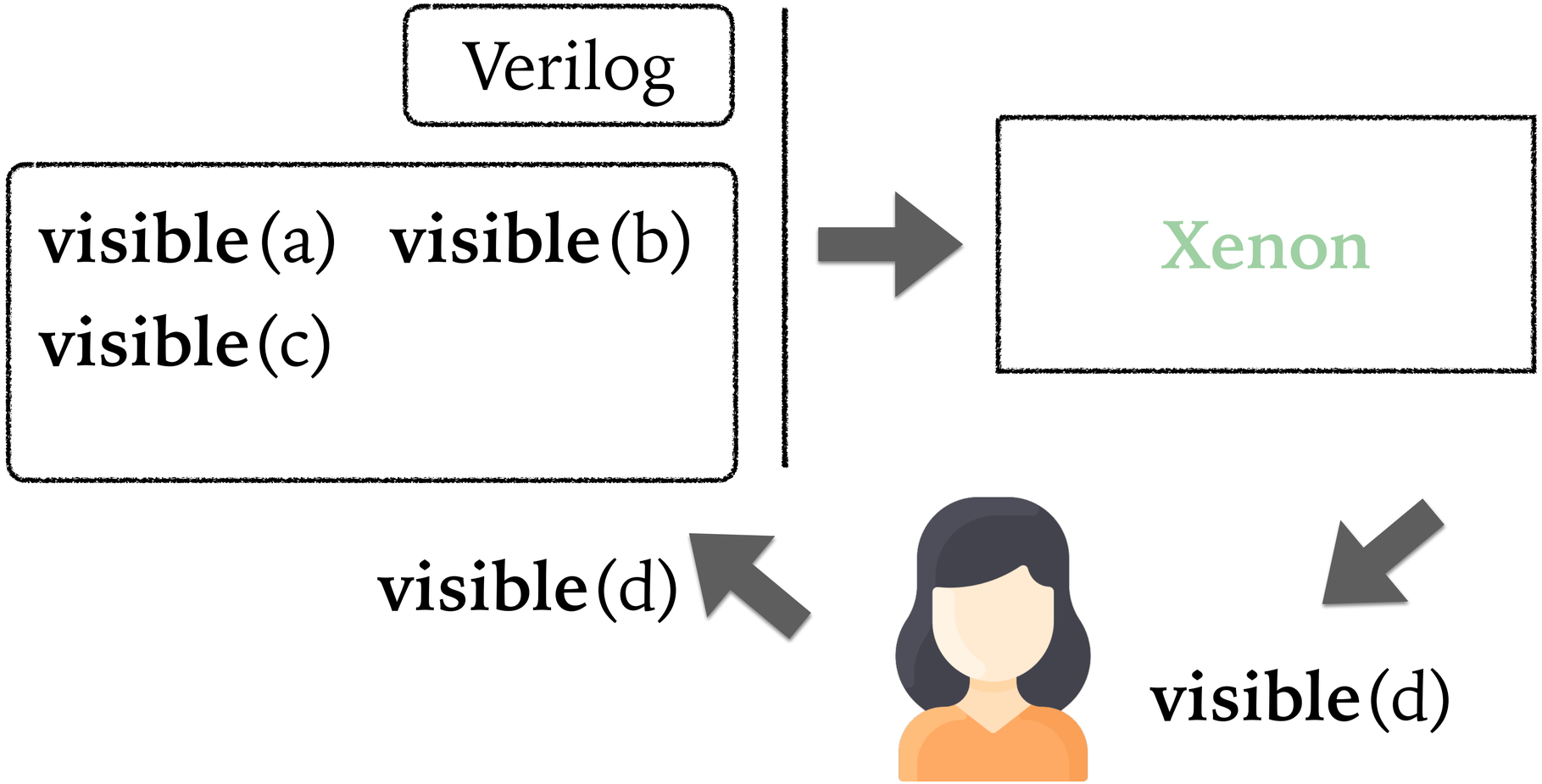
Xenon: Solver-Aided Assumption Synthesis



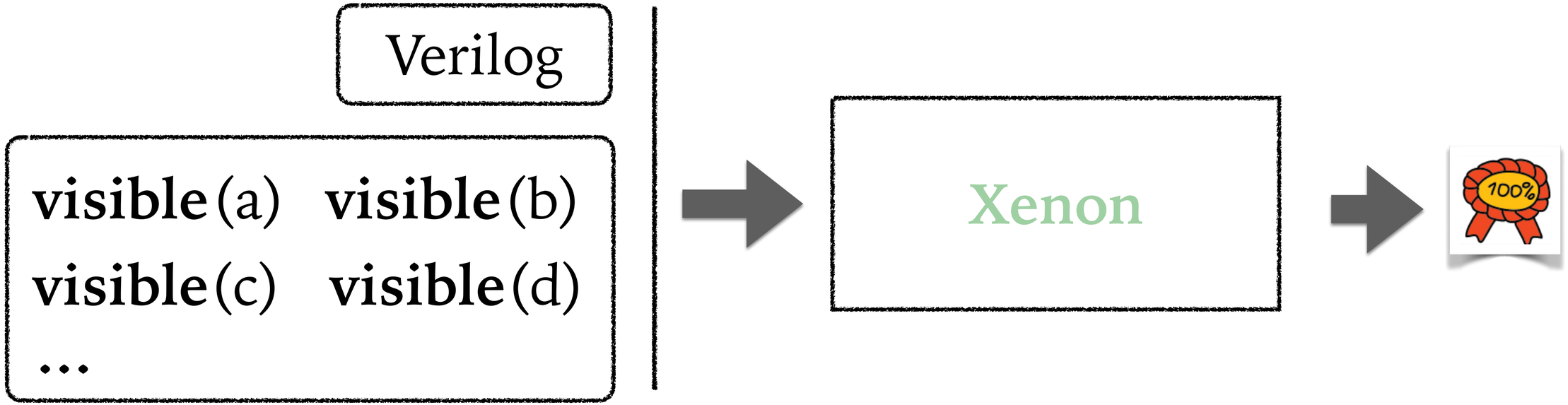
Xenon: Solver-Aided Assumption Synthesis



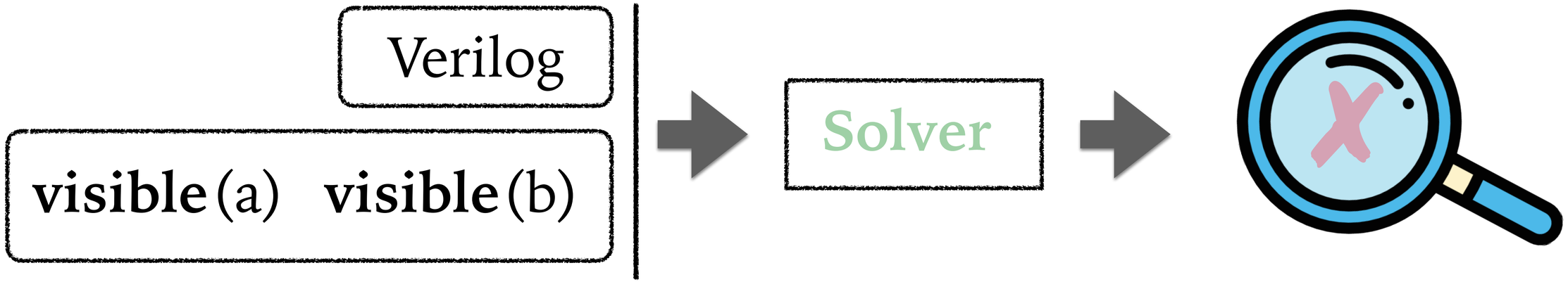
Xenon: Solver-Aided Assumption Synthesis



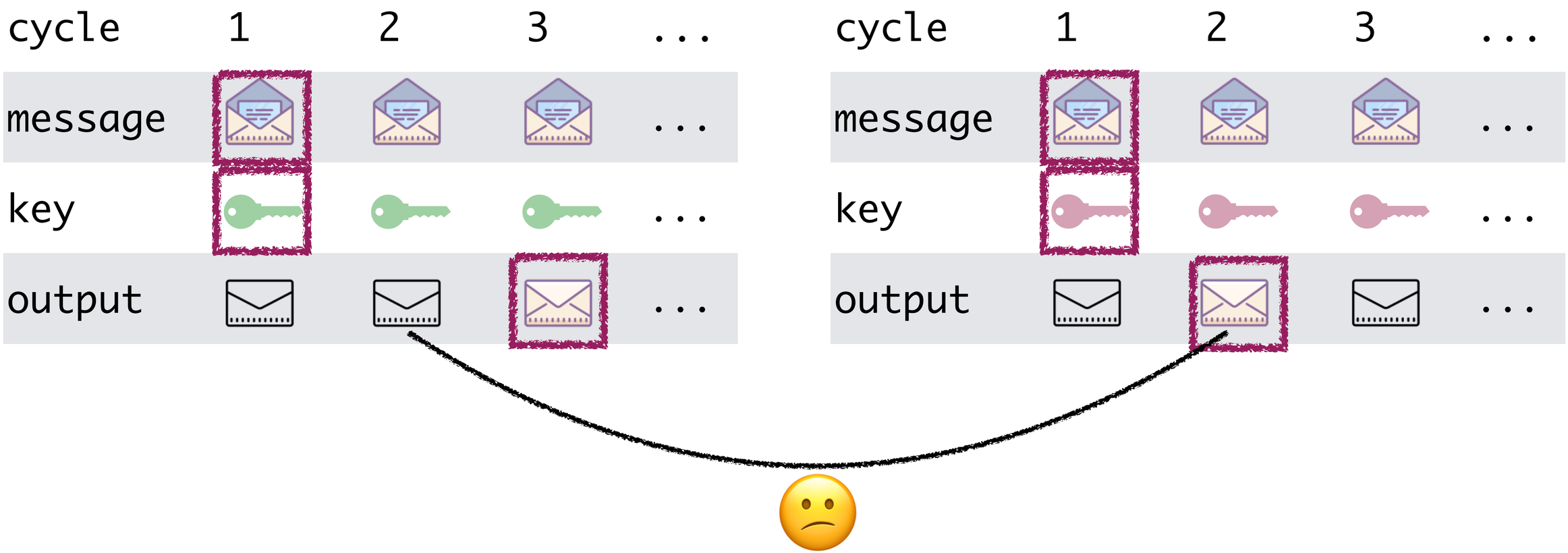
Xenon: Solver-Aided Assumption Synthesis



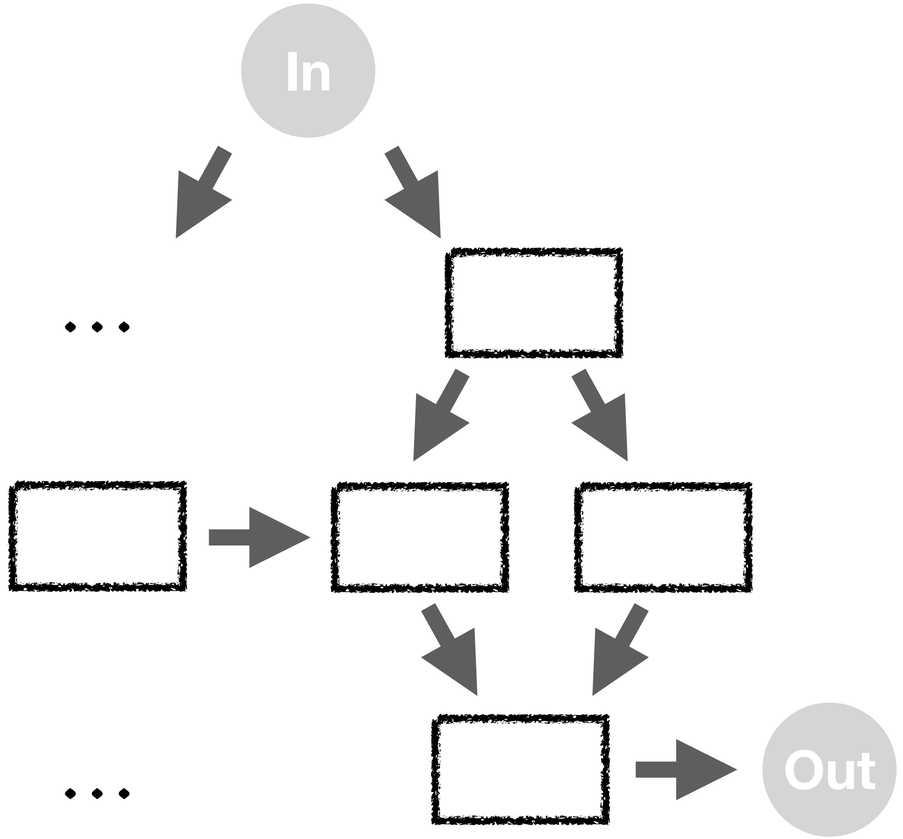
Xenon: Solver-Aided Assumption Synthesis



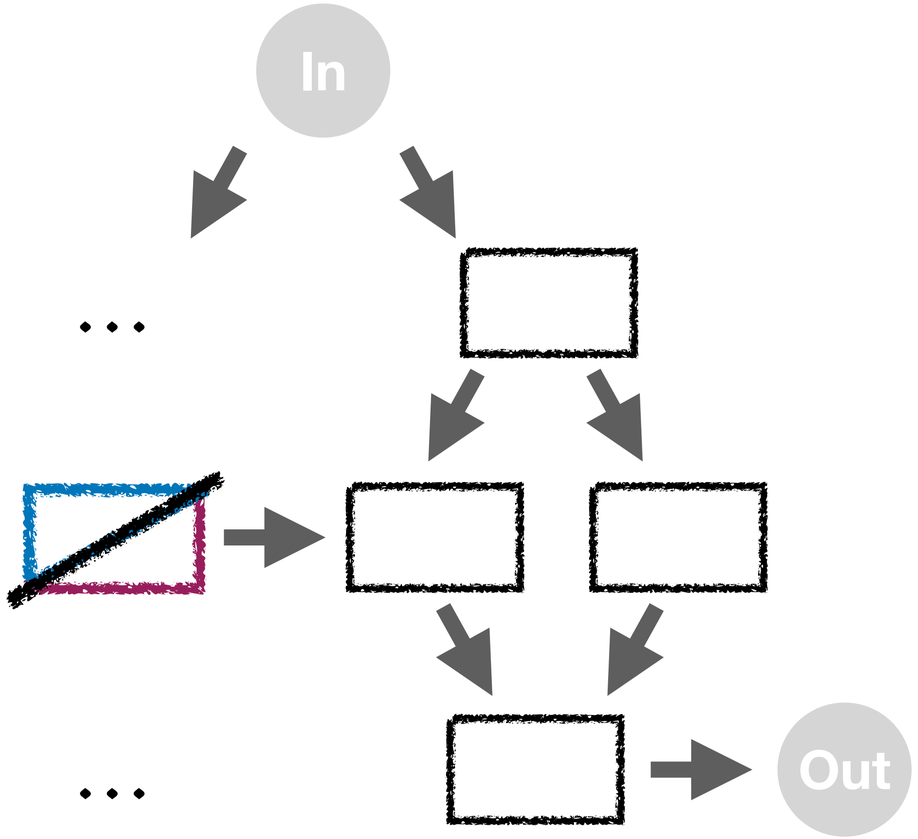
Idea: Trace how variables became *non-ct*



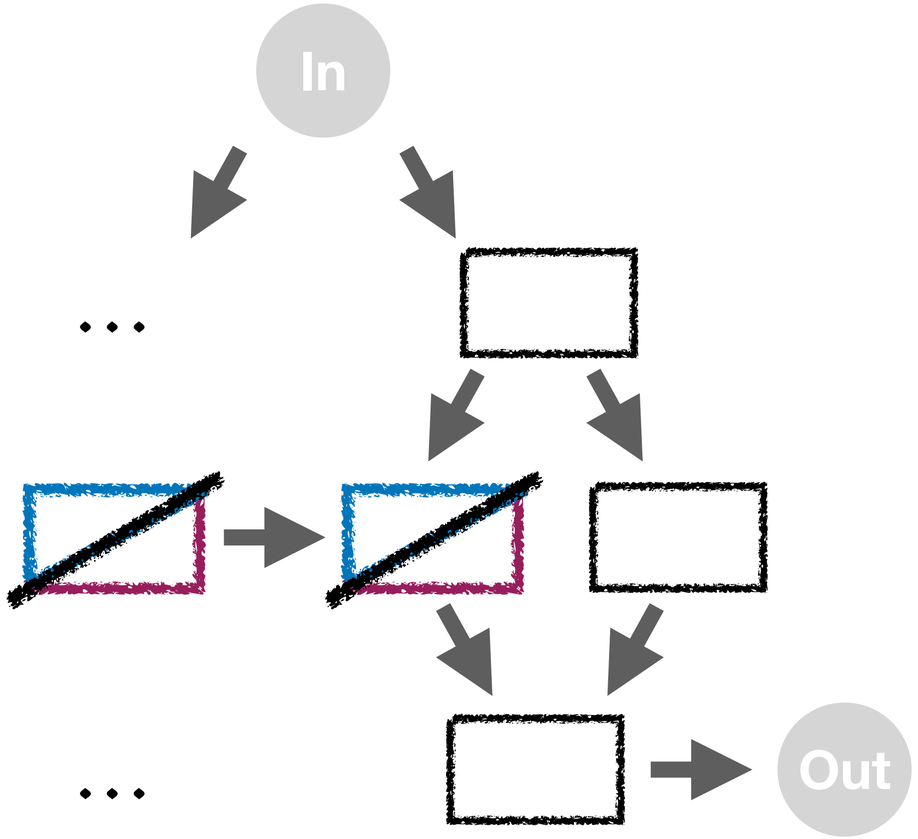
Idea: Trace how variables became *non-ct* 



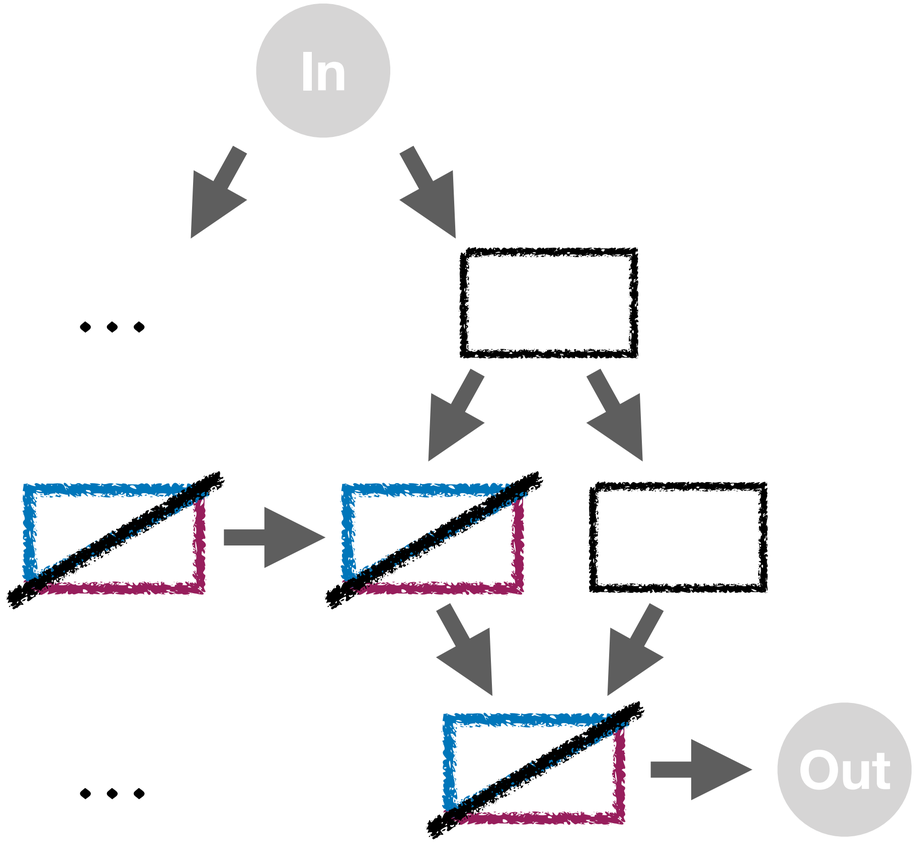
Idea: Trace how variables became *non-ct* 



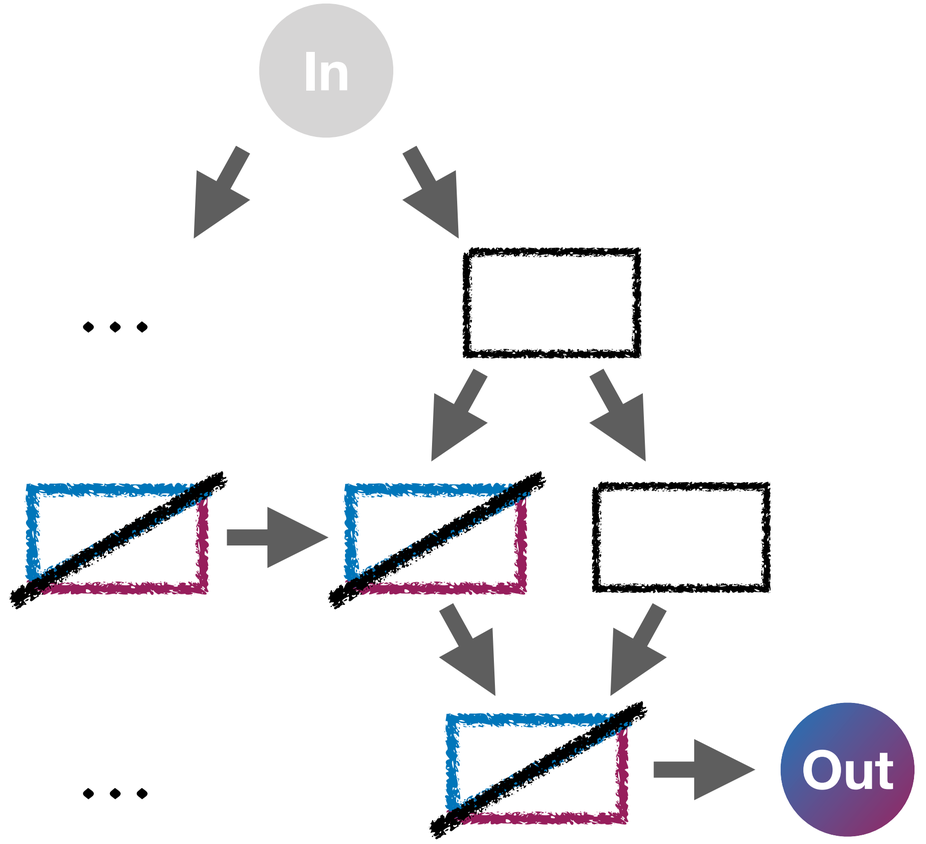
Idea: Trace how variables became *non-ct* 



Idea: Trace how variables became *non-ct* 

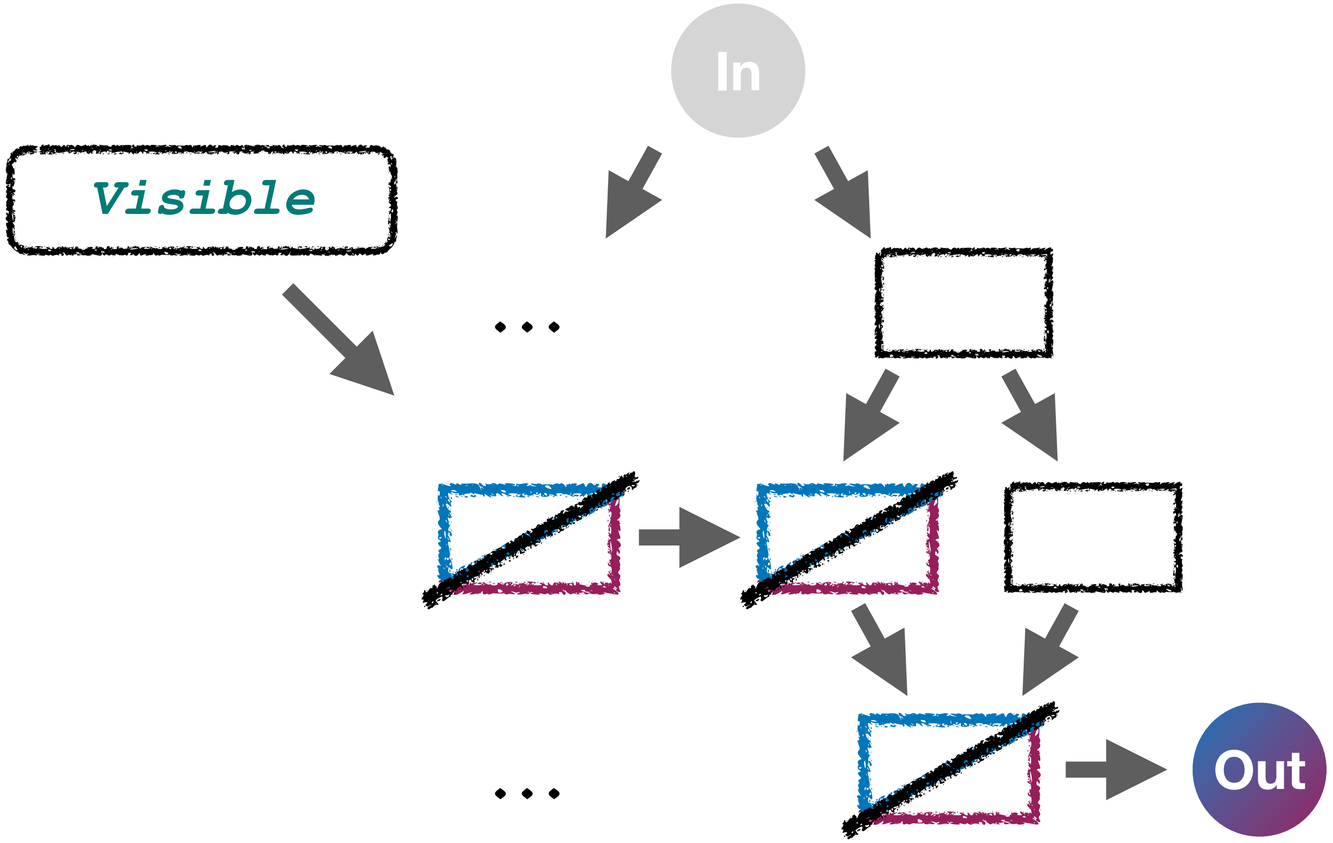


Idea: Trace how variables became *non-ct* 

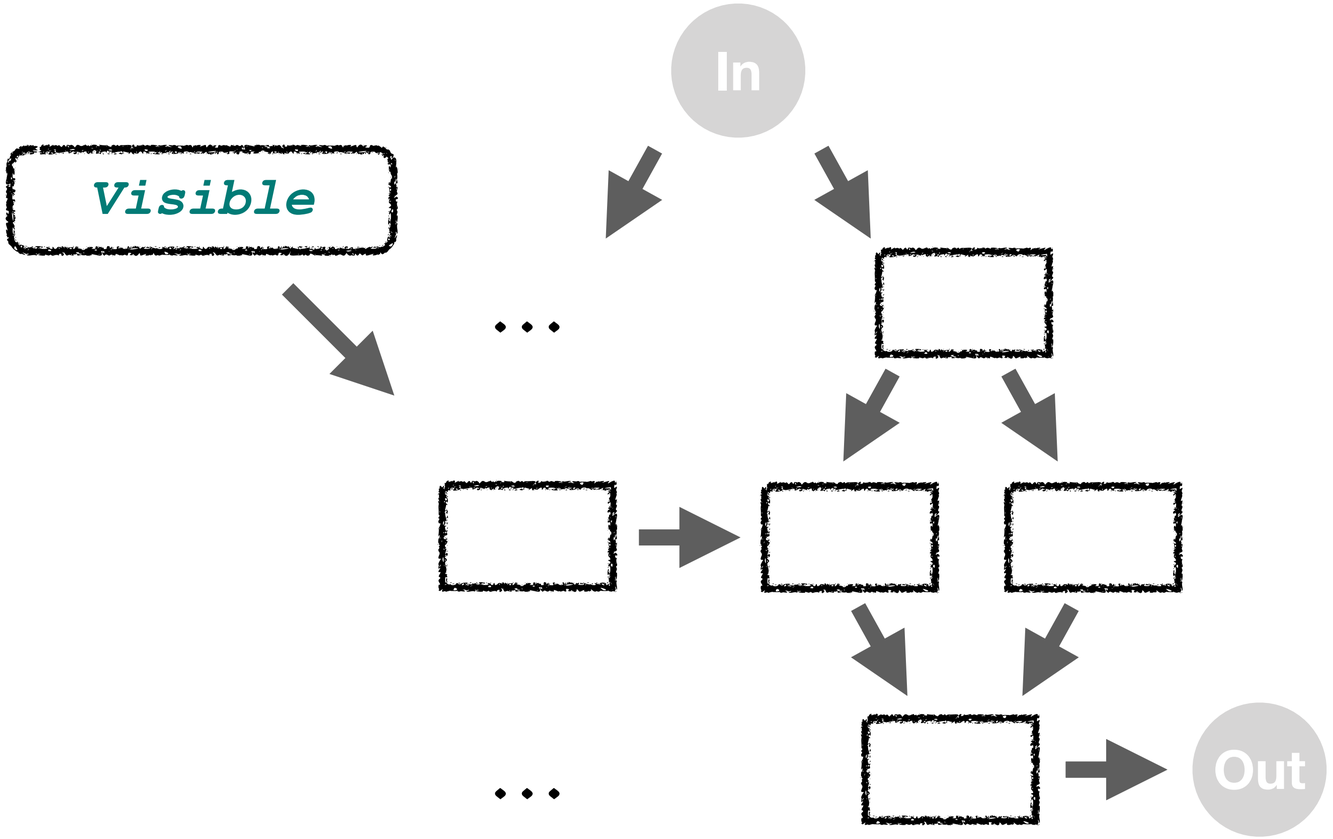


Extract order from failed proof attempt

Fix by adding Assumption



Fix by adding Assumption



Find Assumptions via Optimization Problem

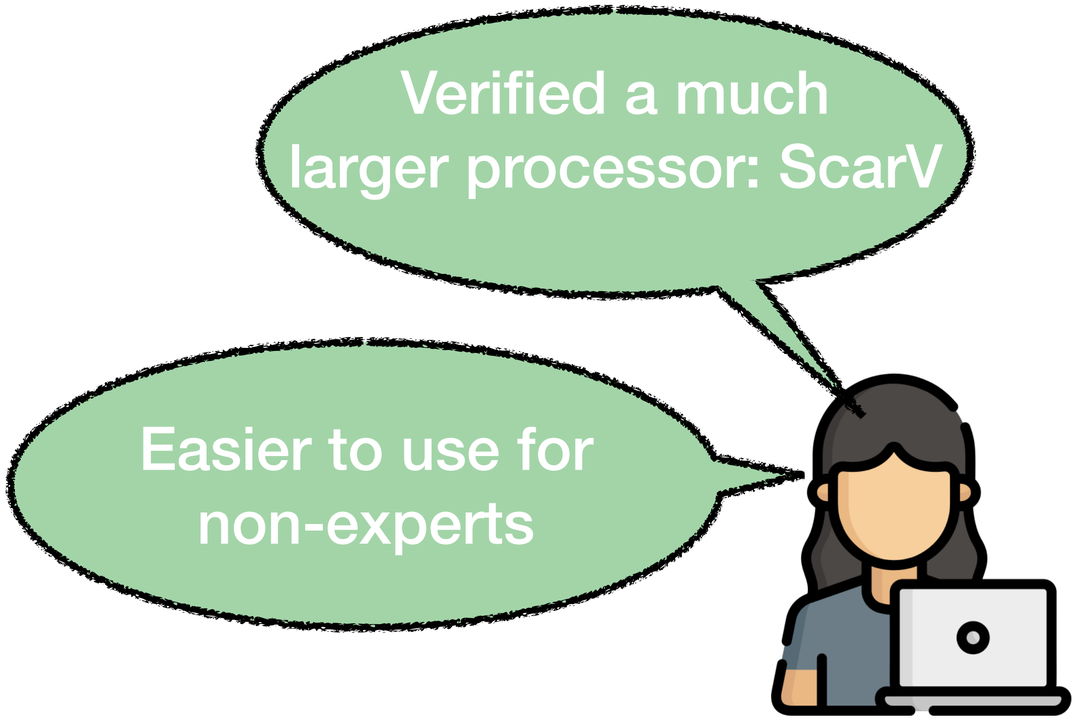
Xenon: Solver-Aided Assumption Synthesis

SCARV-CPU

`https://www.scarv.org/`

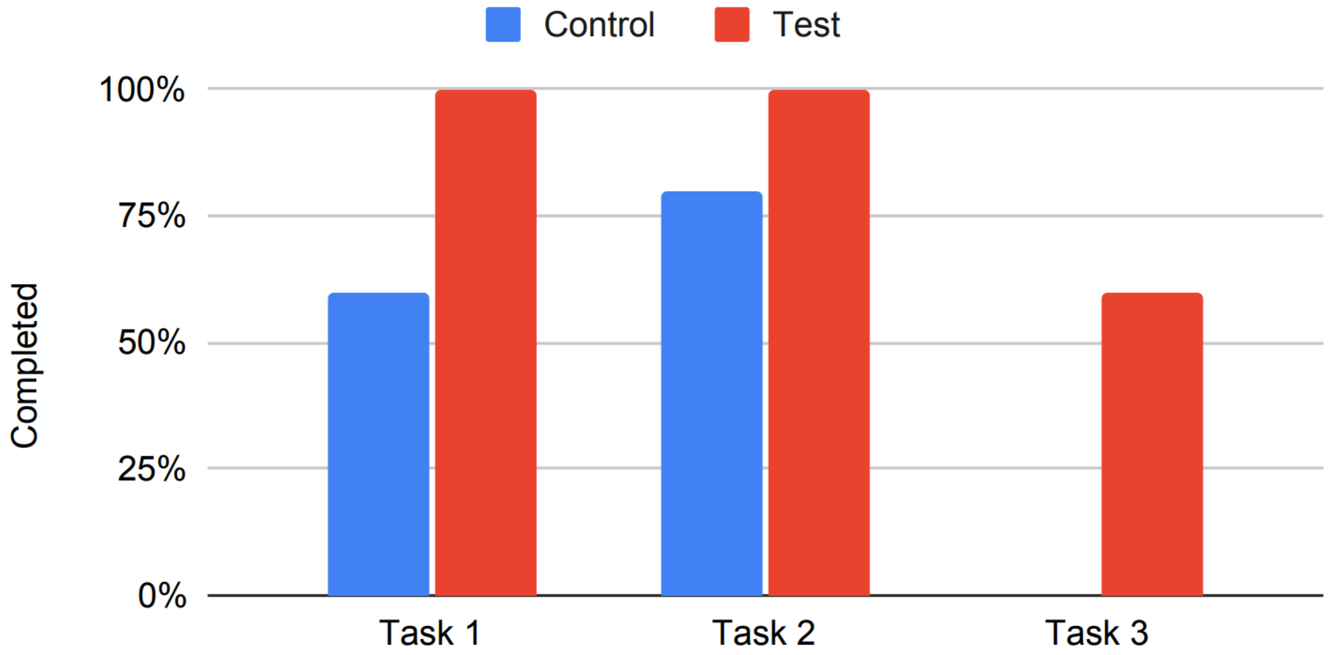
- 5 stage, in-order
- side-channel hardened
- RISC-V 32-bit integer base

Xenon: Solver-Aided Assumption Synthesis



Xenon: **User Study**

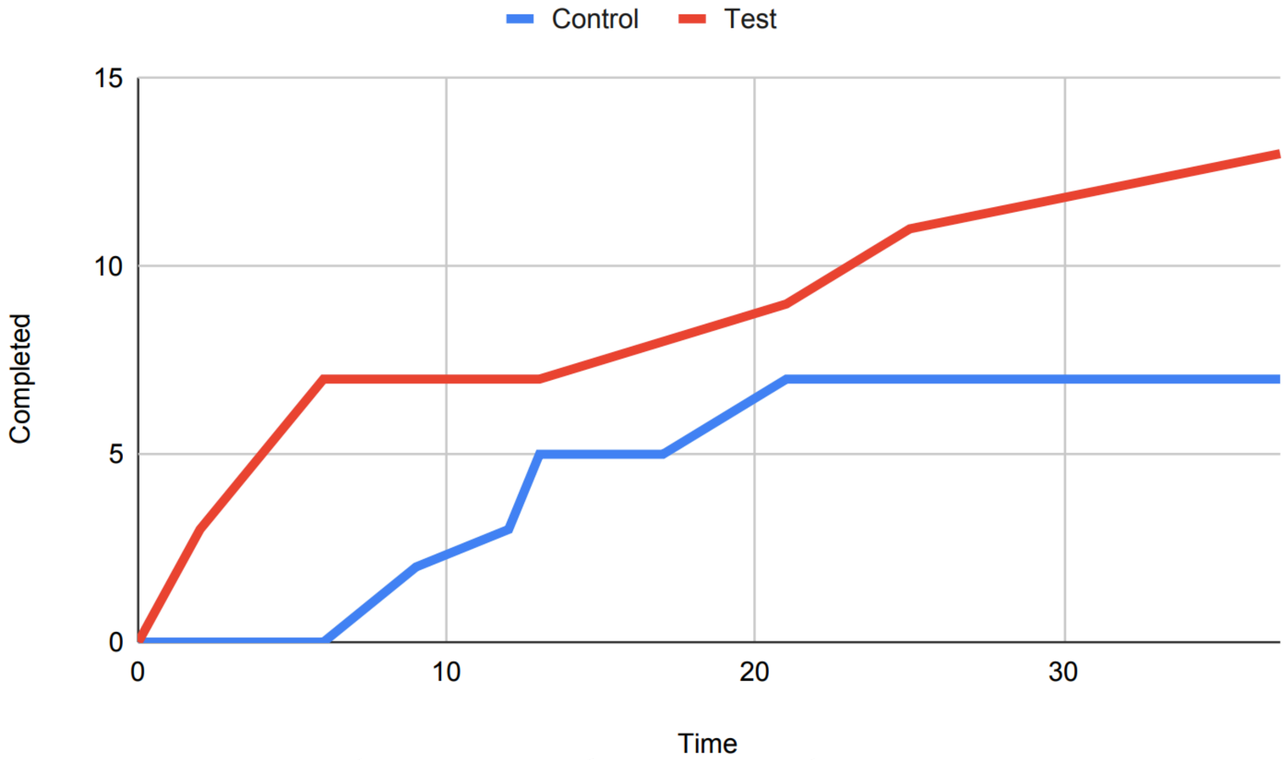
- 3 Tasks: ALU, FPU, and RISC-V core



% of participants who correctly completed

Xenon: User Study

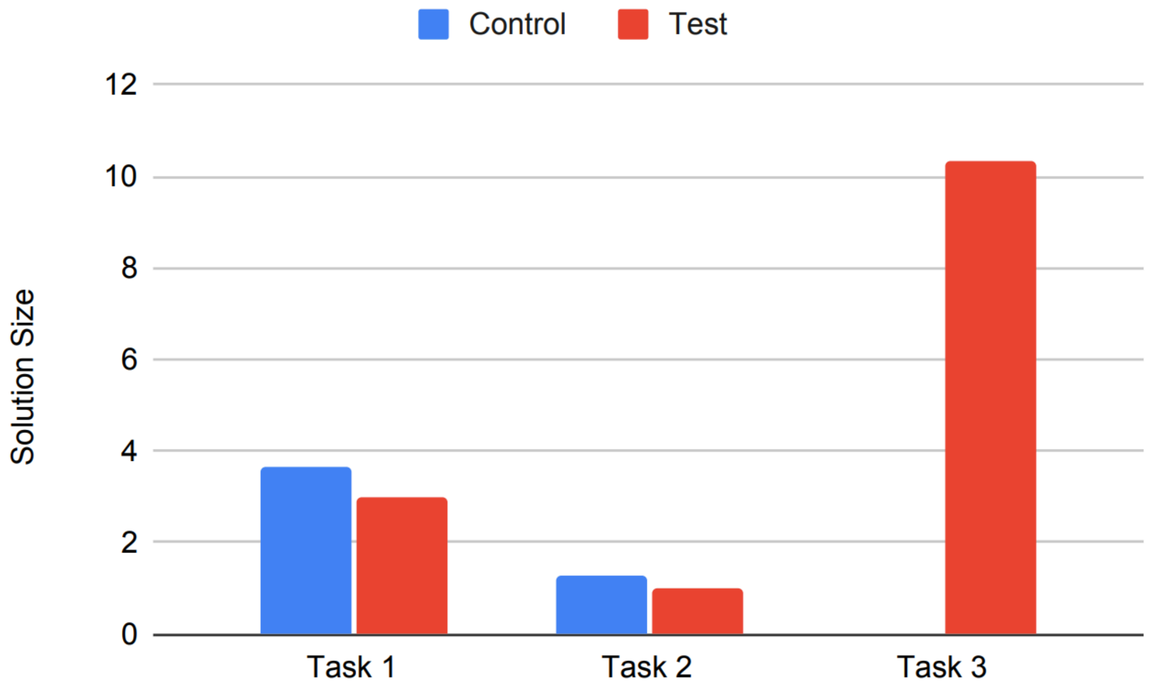
- 3 Tasks: ALU, FPU, and RISC-V core



Completed tasks over time

Xenon: User Study

- 3 Tasks: ALU, FPU, and RISC-V core



Solution sizes

My Journey

[1] Iodine: What's timing in HW. **Usenix Security**'19.

[2] Xenon: Help non-experts find assumptions. **CCS**'21.

[3] LeaVe: Assumptions at Software level. **CCS**'23. **Distinguished paper**.

My Journey

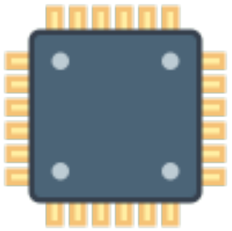
[1] Iodine: What's timing in HW. **Usenix Security**'19.

[2] Xenon: Help non-experts find assumptions. **CCS**'21.

[3] LeaVe: Assumptions at Software level. **CCS**'23.
Distinguished paper.

Problem: Assumptions about internal processor state

Problem: Assumptions about internal processor state

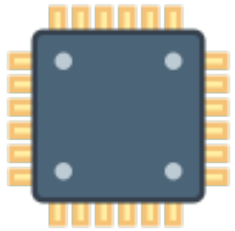


hardware

software

Complicated: How to write software that satisfies assumptions?

LeaVe: Represent assumptions directly in software

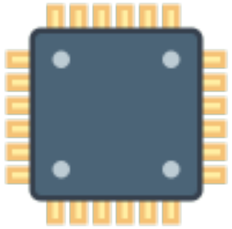


hardware

software

…and formally verify that under the assumptions hardware doesn't leak

software leakage description



Verilog

LeaVe

proof

counterexample

: If no leakage in software → Guaranteed no leakage in hardware

Example: Simple Processor

- Single register
- Instructions:

ADD imm

MUL imm

CLR

Example: Simple Processor

ADD 1 MUL 2 CLR

0

Example: Simple Processor

ADD 1 MUL 2 CLR

0

Example: Simple Processor

ADD 1 MUL 2 CLR

Example: Simple Processor

ADD 1 MUL 2 CLR

1

Example: Simple Processor

ADD 1 MUL 2 CLR

2

Example: Simple Processor

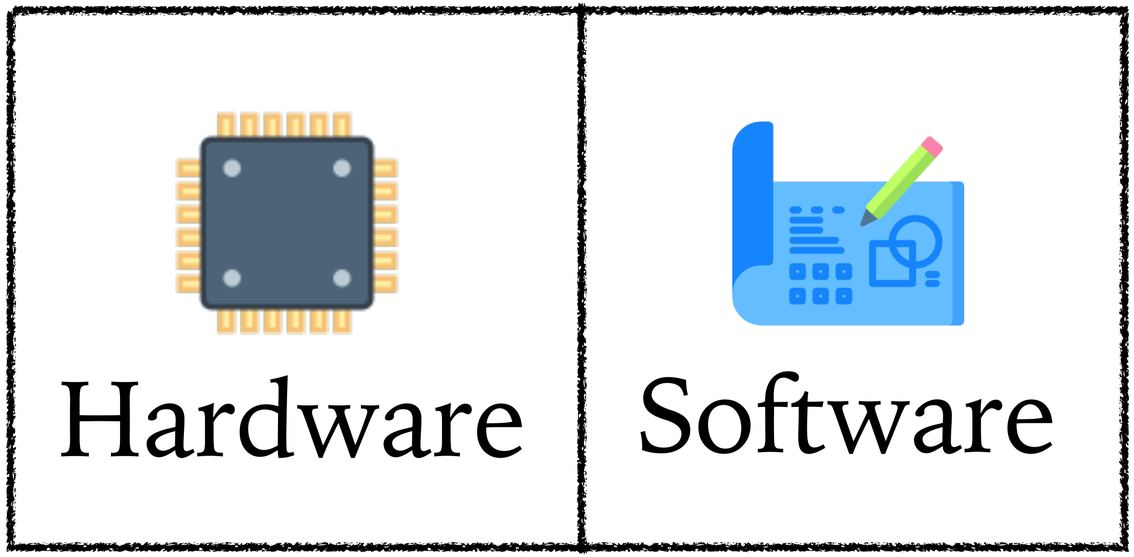
ADD 1 MUL 2 CLR

2

Example: Simple Processor

ADD 1 MUL 2 CLR

0



Software level specification is much simpler

Software

```
1   wire [31:0] instr = imem[pc];
2   assign op = instr[7:0];
3   assign imm = instr[31:8];

4   always @(posedge clk) begin
5     pc <= pc + 1;
6   end

7 always @ (posedge clk) begin
8   case(op)
9         ‘ADD : register <= register + imm;
10        ‘MUL : register <= register * imm;
11        ‘CLR : register <= 0;
18 end
```

Software

```
1   wire [31:0] instr = imem[pc];
2   assign op = instr[7:0];
3   assign imm = instr[31:8];

4   always @(posedge clk) begin
5     pc <= pc + 1;
6   end

7 always @ (posedge clk) begin
8   case(op)
9         ‘ADD : register <= register + imm;
10        ‘MUL : register <= register * imm;
11        ‘CLR : register <= 0;
18 end
```

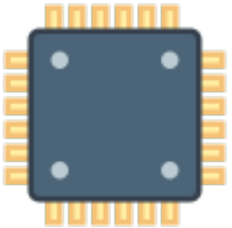
Software

```
1   wire [31:0] instr = imem[pc];
2   assign op = instr[7:0];
3   assign imm = instr[31:8];

4   always @(posedge clk) begin
5     pc <= pc + 1;
6   end

7 always @ (posedge clk) begin
8   case(op)
9         ‘ADD : register <= register + imm;
10        ‘MUL : register <= register * imm;
11        ‘CLR : register <= 0;
18 end
```

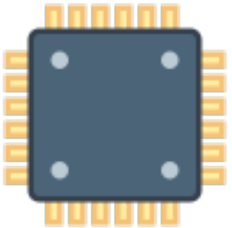
Software

```
1   wire [31:0] instr = imem[pc];
2   assign op = instr[7:0];
3   assign imm = instr[31:8];

4   always @(posedge clk) begin
5     pc <= pc + 1;
6   end

7 always @ (posedge clk) begin
8   case(op)
9         ‘ADD : register <= register + imm;
10        ‘MUL : register <= register * imm;
11        ‘CLR : register <= 0;
18 end
```

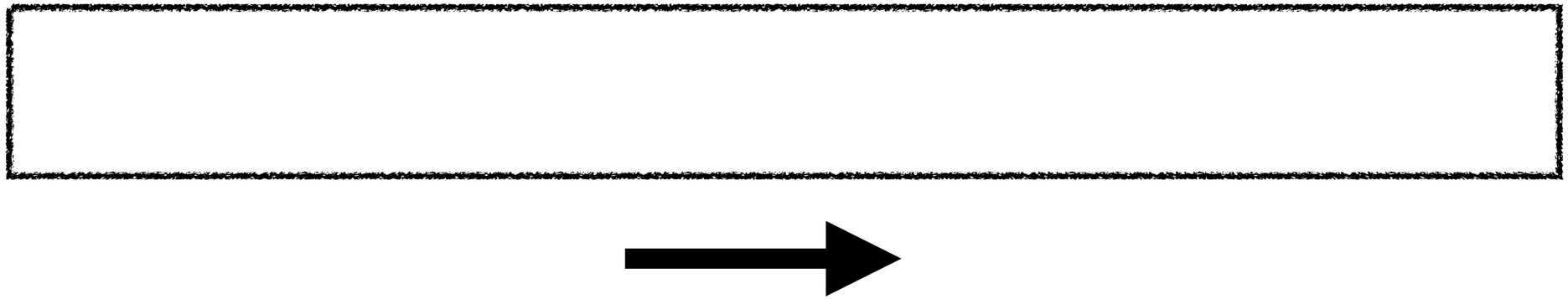
Hardware



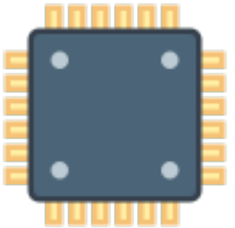
Hardware



Fetch Execute Write-Back



Hardware

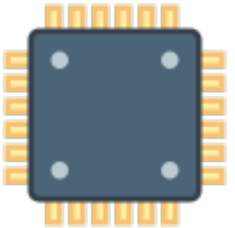


ADD 1 MUL 2 CLR ADD 3

Fetch Execute Write-Back

0

Hardware



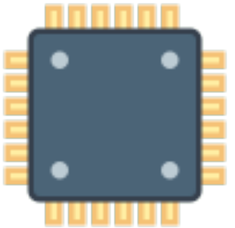
ADD 1 MUL 2 CLR ADD 3

Fetch Execute Write-Back

ADD 1	No-op	No-op

0

Hardware



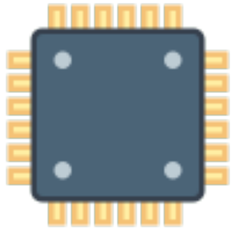
ADD 1 MUL 2 CLR ADD 3

Fetch Execute Write-Back

MUL 2	ADD 1	No-op

0

Hardware

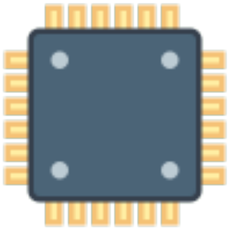


ADD 1 MUL 2 CLR ADD 3

Fetch Execute Write-Back

0

Hardware



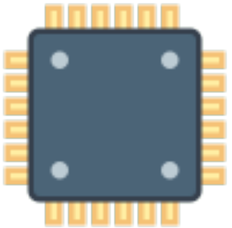
ADD 1 MUL 2 CLR ADD 3

Fetch Execute Write-Back

ADD 3	CLR	MUL 2

1

Hardware



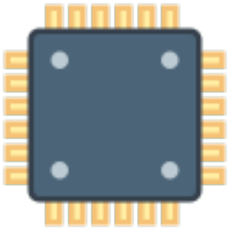
ADD 1 MUL 2 CLR ADD 3

Fetch Execute Write-Back

… ADD 3 CLR

2

Hardware



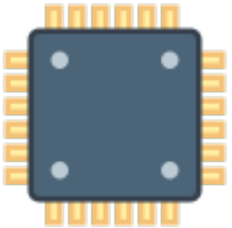
ADD 1 MUL 2 CLR ADD 3

Fetch Execute Write-Back

… … ADD 3

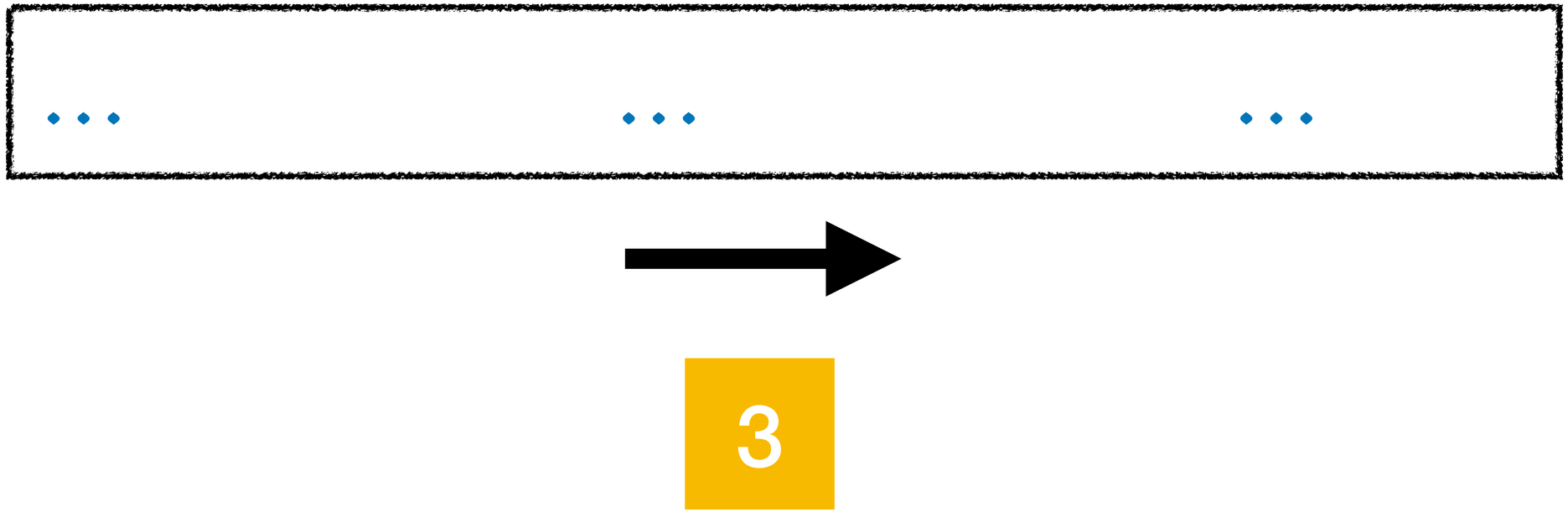
0

Hardware



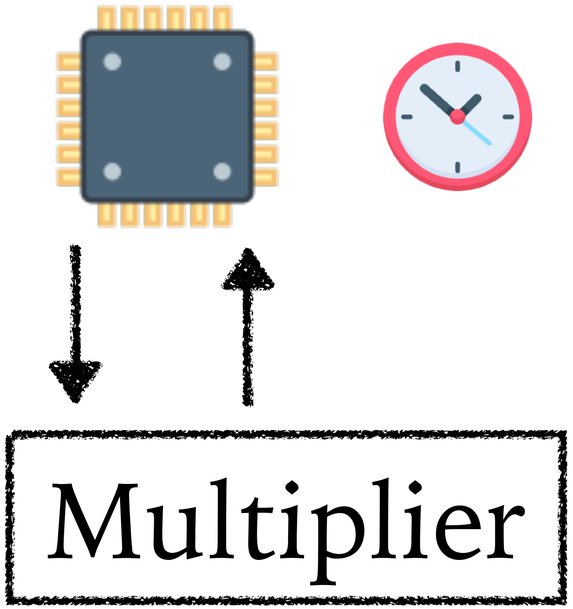
ADD 1 MUL 2 CLR ADD 3

Fetch Execute Write-Back



3

Hardware: Leaky Multiplier



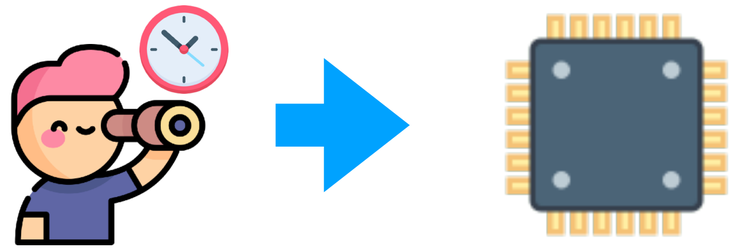
Take multiple cycles

Fast path on 0/1 for imm

Time depends on register

How to describe hardware leakage at software level?

The attacker can see execution time

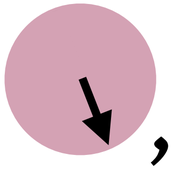


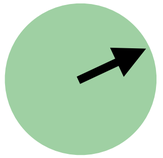
attacker processor

Which programs can an attacker distinguish?

Which programs can an attacker distinguish?

ADD 2 MUL 15 ADD 2 ADD 2

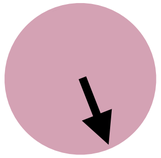
Left uses MUL 

, right doesn't 

Which programs can an attacker distinguish?

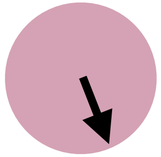
ADD 2 MUL 1 ADD 2 MUL 15

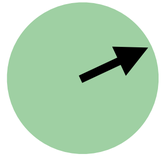
Left takes fast path 

, right doesn't 

Which programs can an attacker distinguish?

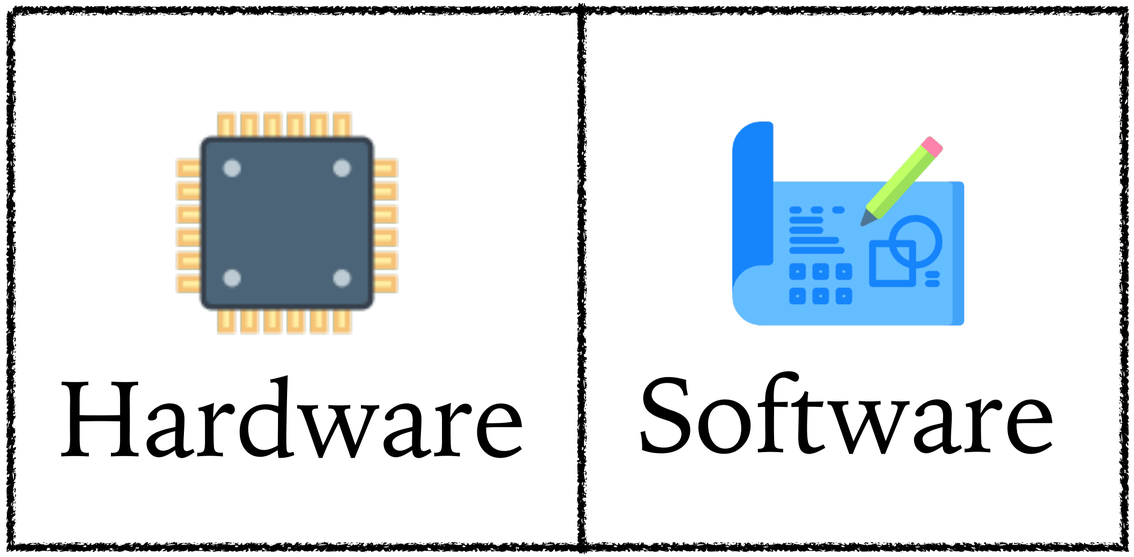
ADD 10 MUL 2 ADD 2 MUL 2

Left register value is higher 

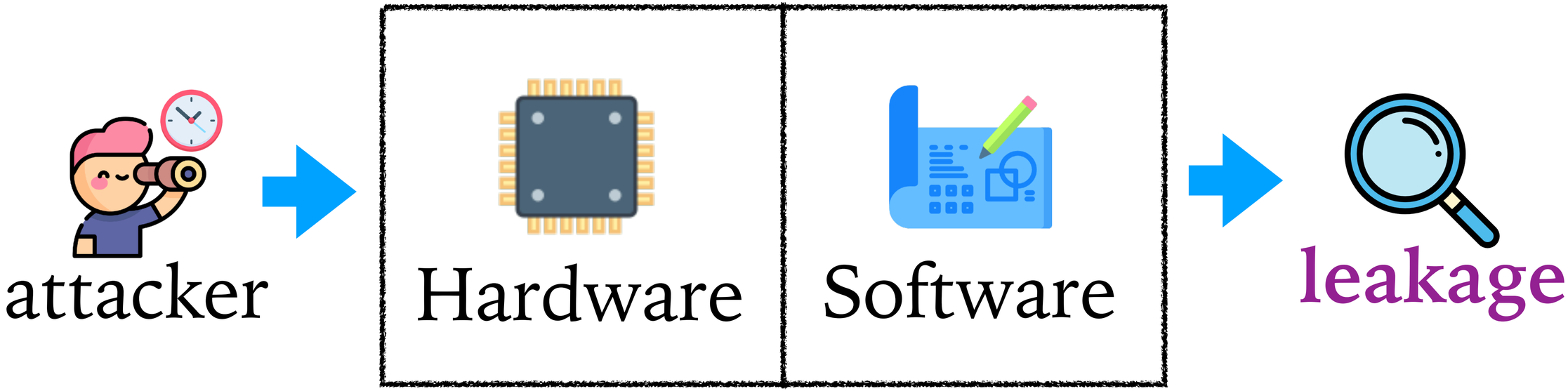
than right 

Idea: Describe leakage via leakage monitor

Idea: Describe leakage via leakage monitor



Idea: Describe leakage via leakage monitor



Make sure the hardware doesn't leak more than specified by the monitor

LeaVe: Capture leakage via leakage monitor

Defining the leakage monitor

Checking the leakage monitor

Verifying the leakage monitor

} done by LeaVe

Using the leakage monitor

Defining the **leakage monitor**

{ , , }

Defining the **leakage monitor**

- Is the operation MUL?

{isMul, , }

Defining the **leakage monitor**

{isMul, fast, }

- Is the operation MUL?
- Did we take the fast-path?

Defining the **leakage monitor**

{isMul, fast, register}

- Is the operation MUL?
- Did we take the fast-path?
- Register value.

Defining the **leakage monitor**

{isMul, fast, register}	{F, -,-}	{T, F, 2}	{T, T, 2}
	ADD 2	MUL 3	MUL 1

LeaVe: Capture leakage via leakage monitor

Defining the leakage monitor

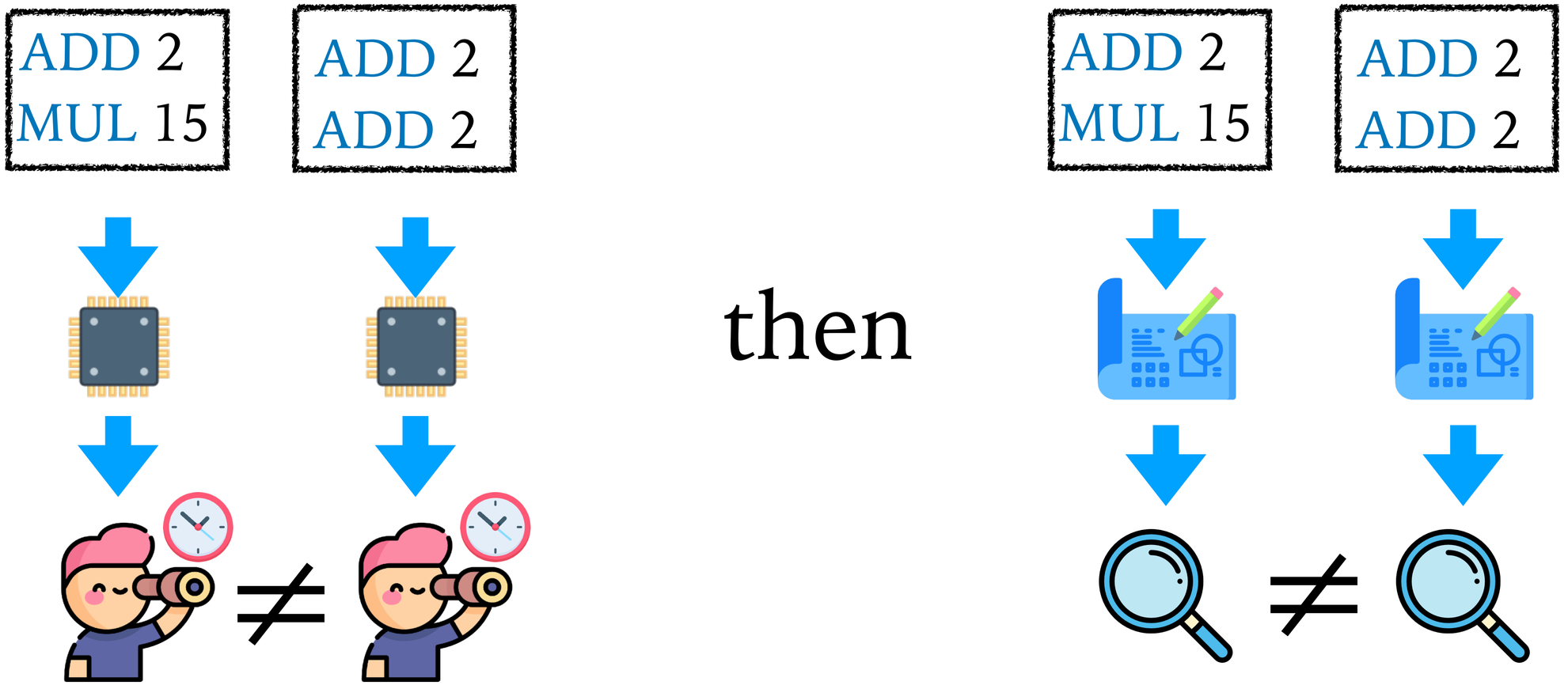
Checking the leakage monitor

Verifying the leakage monitor

} done by LeaVe

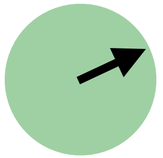
Using the leakage monitor

Checking the leakage monitor



Checking the leakage monitor

Left uses MUL 

, right doesn't 

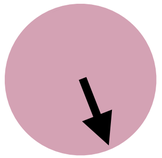
≠

{F, -,-} {T, F, 2} {F, -,-} {F, -,-}

ADD 2 MUL 15 ADD 2 ADD 2

Checking the leakage monitor

Left takes <u>fast path</u> 

, right doesn't 

$\neq$

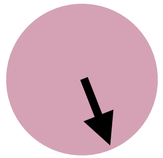
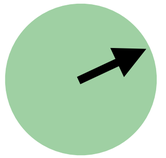
{F, -,-} {T, T, 2}

ADD 2 MUL 1

{F, -,-} {T, F, 2}

ADD 2 MUL 15

Checking the leakage monitor

Left register value is higher  than right 

≠

{F, -,-}	{T, F, 10}	{F, -,-}	{T, F, 2}
ADD 10	MUL 2	ADD 2	MUL 2

LeaVe: Capture leakage via leakage monitor

Defining the leakage monitor

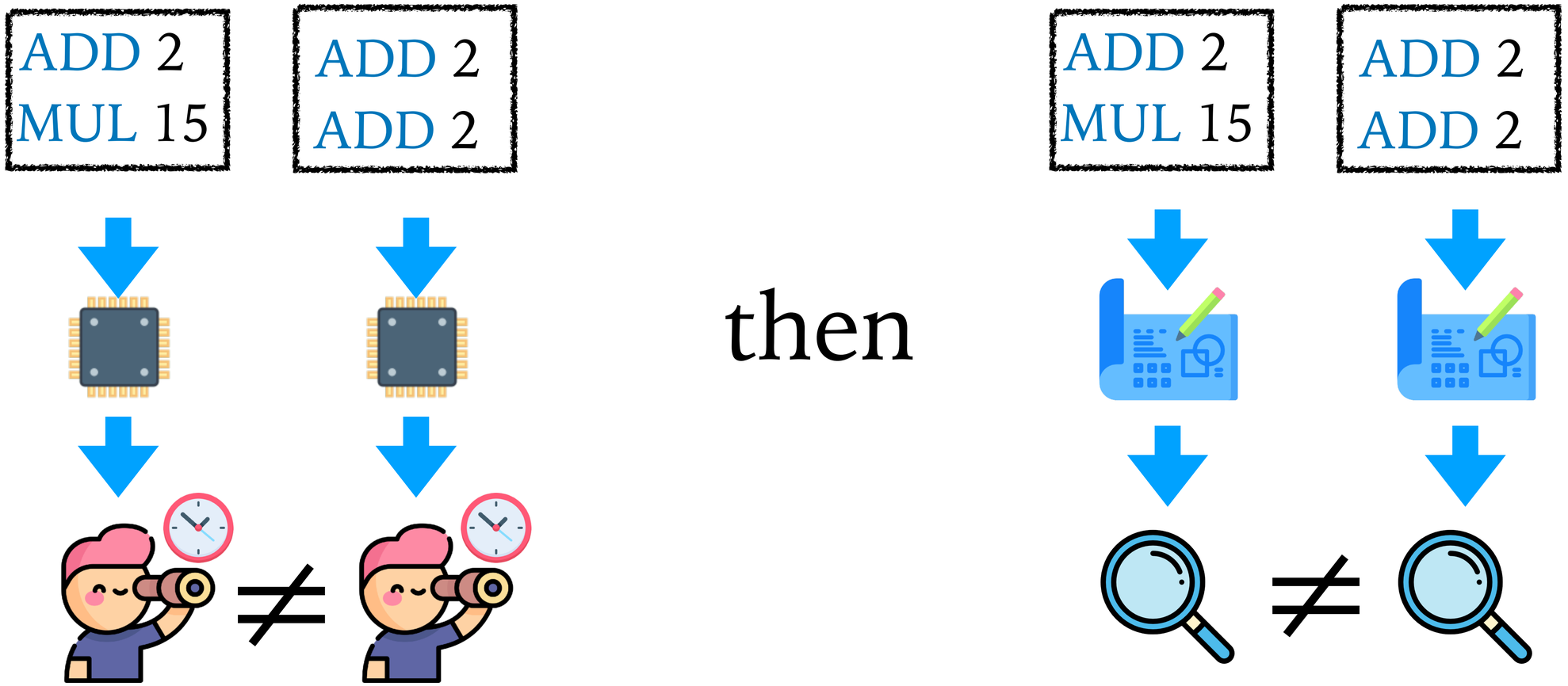
Checking the leakage monitor

Verifying the leakage monitor

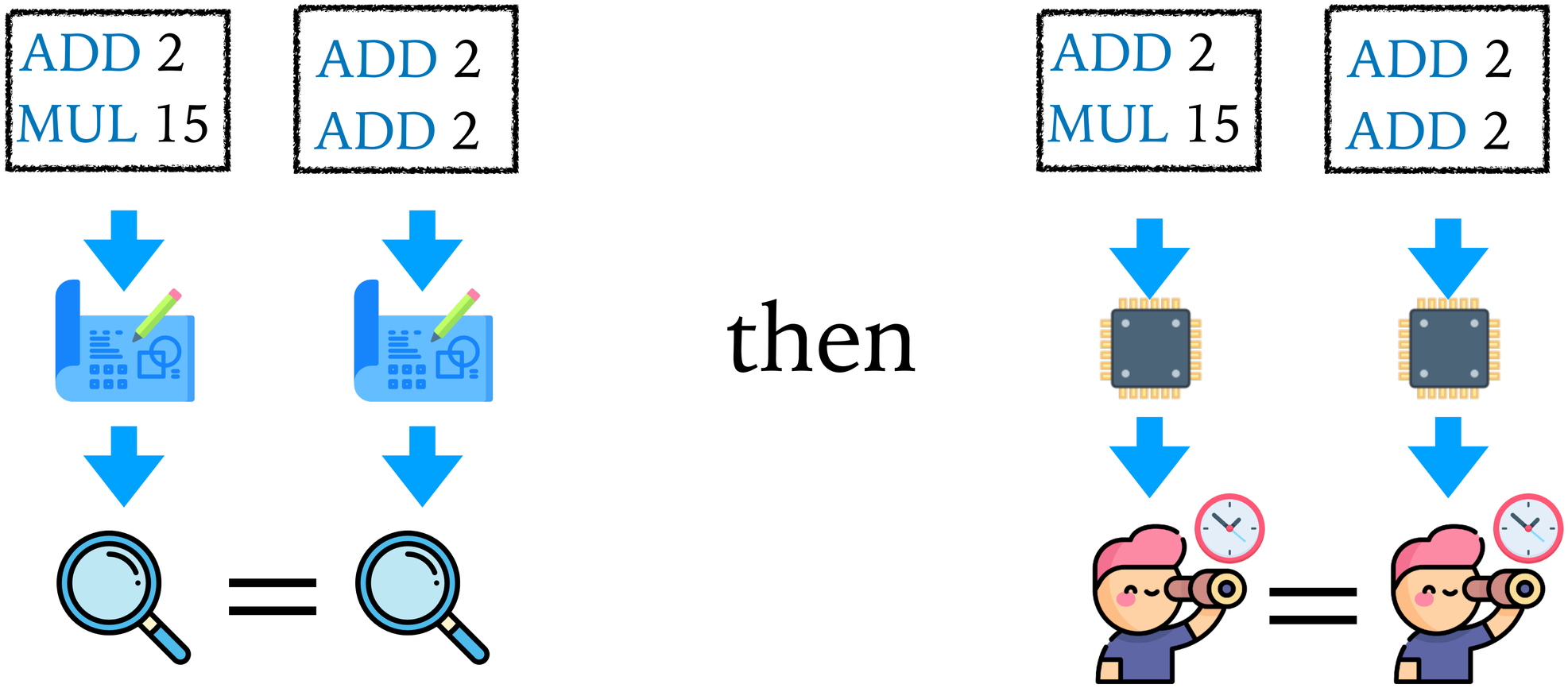
} done by LeaVe

Using the leakage monitor

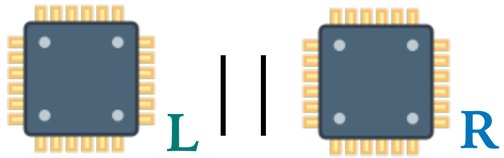
Verifying the leakage monitor



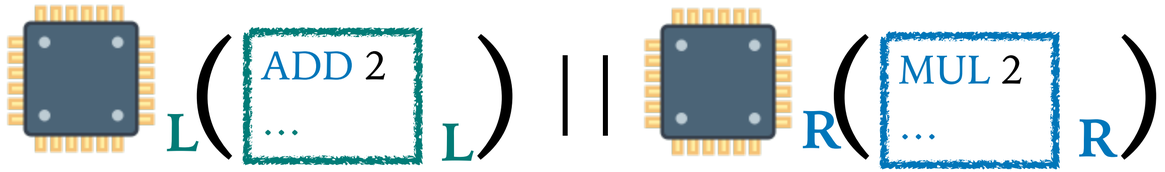
Verifying the **leakage monitor**



Verifying the leakage monitor

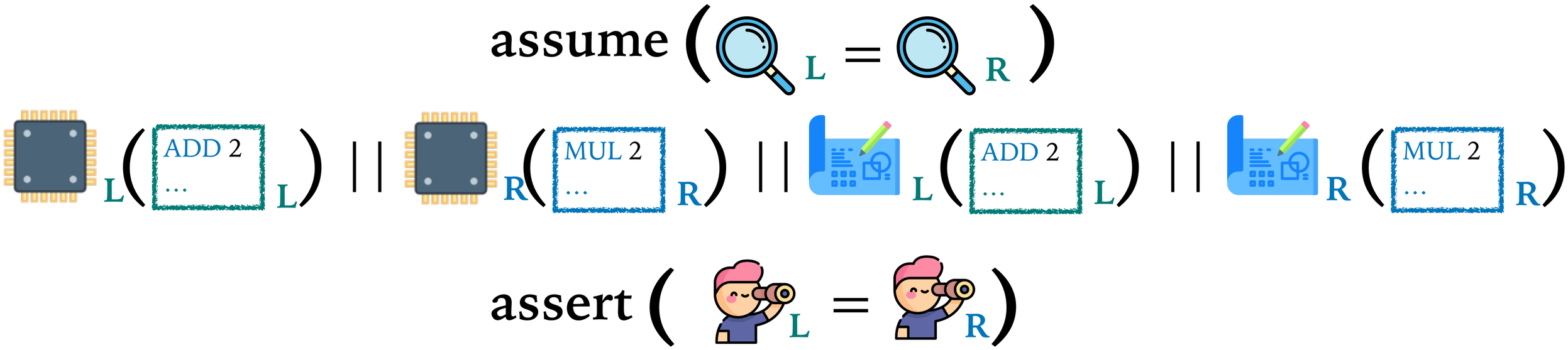


Verifying the leakage monitor



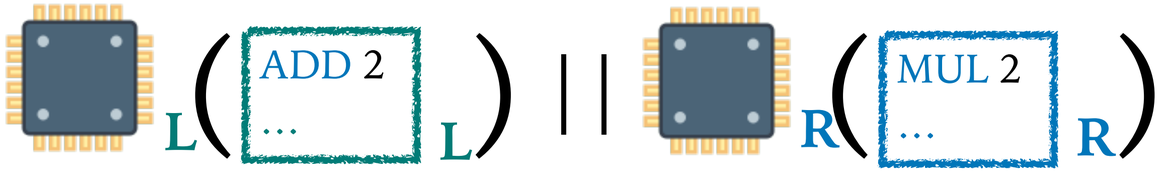
Verifying the leakage monitor

Verifying the leakage monitor



2+2 copies. Doesn't scale!

Verifying the **leakage monitor**



Only 2 copies. Scales!

* assuming no functional bugs

LeaVe: Capture leakage via leakage monitor

Defining the leakage monitor

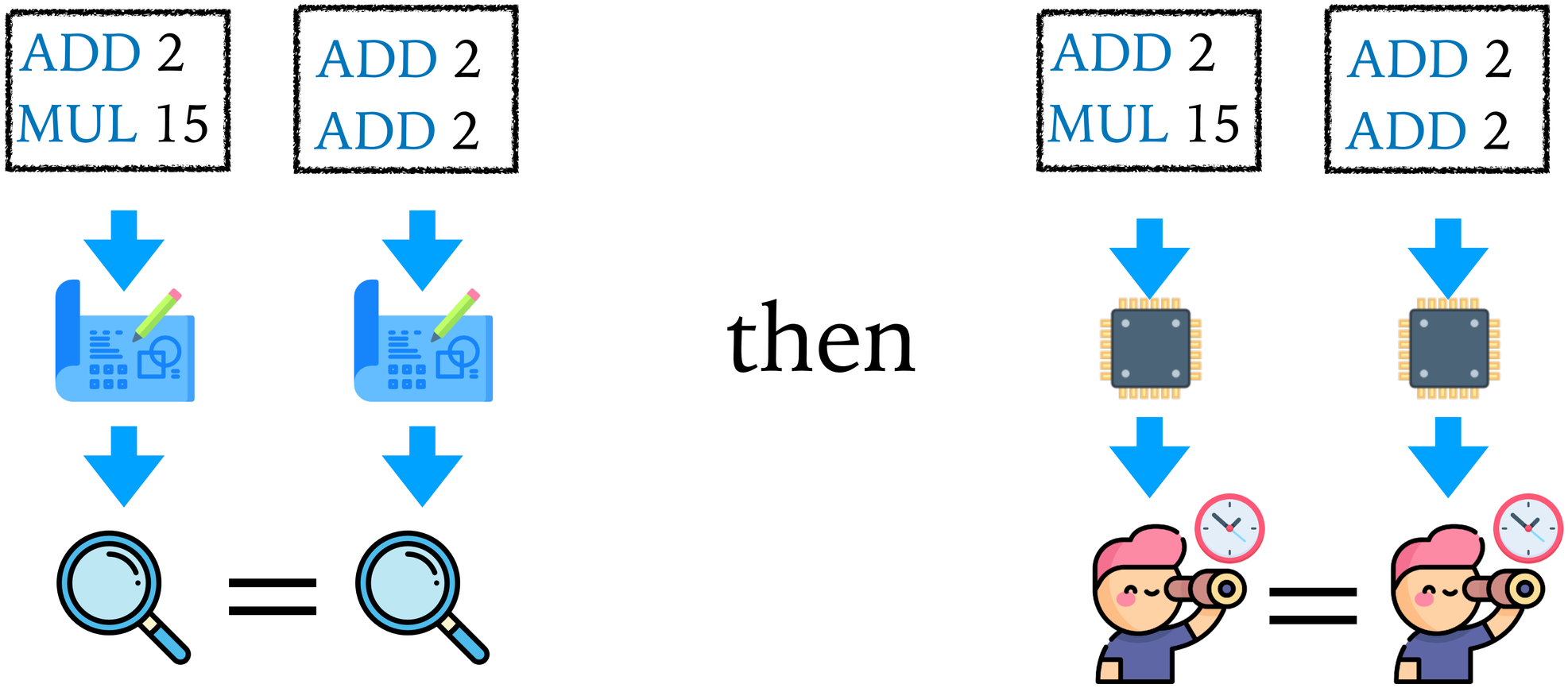
Checking the leakage monitor

Verifying the leakage monitor

} done by LeaVe

Using the leakage monitor

Using the leakage monitor



Evaluating LeaVe

Formally verified how to avoid leakage in software on real processors!

Processor	Leakage Description	Checking Time (min)
DarkRISC 2 stage	I	7.2
DarkRISC 3 stage	I	11.1
Sodor 2 stage	I + B	97.8
Ibex-small	I + B	118.7
Ibex-slow-mult	I + B + O	110.5
Ibex-cache	I + B + M	139.8

: Instructions

: Branch

: Address

: Operand

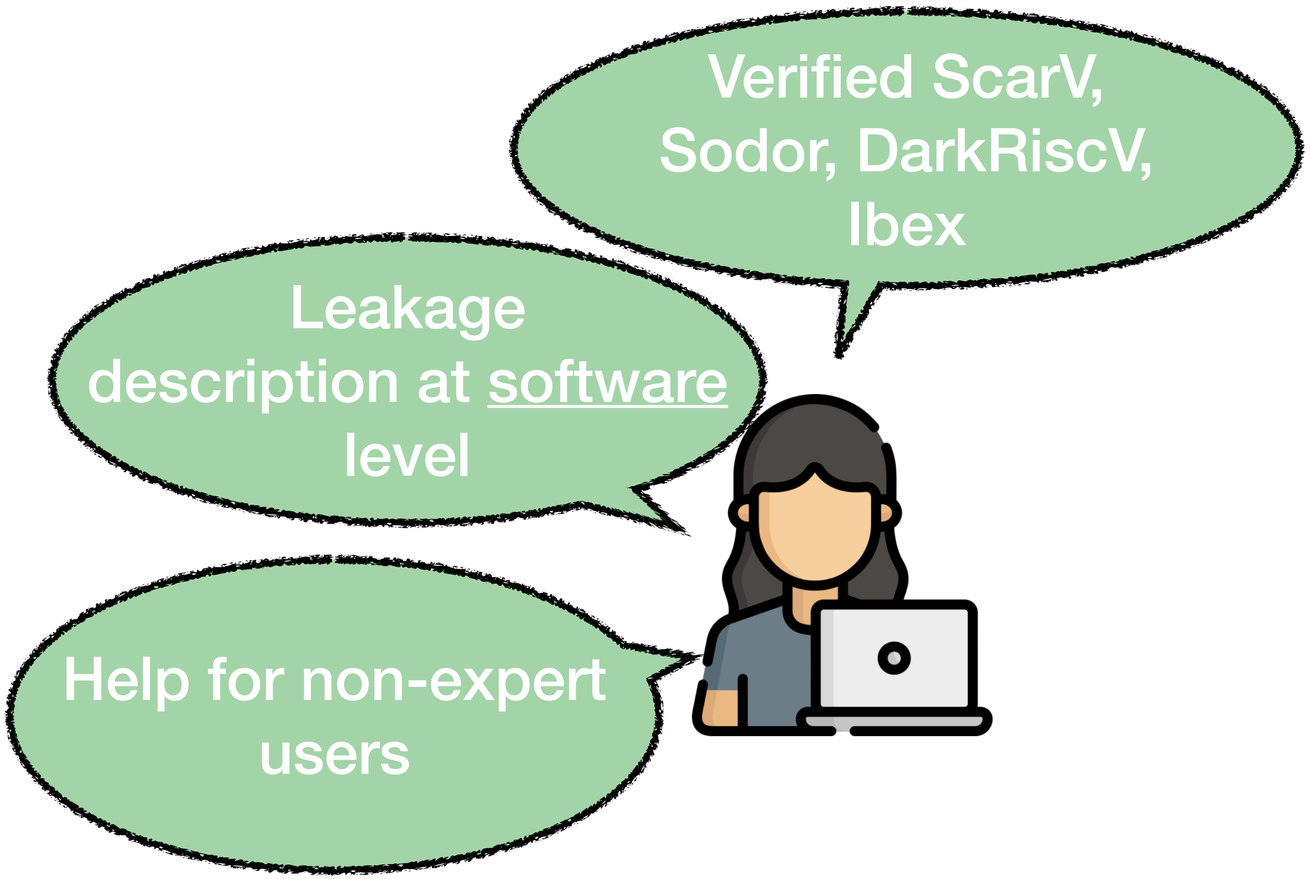
My Journey

[1] Iodine: What's timing in HW. **Usenix Security**'19.

[2] Xenon: Help non-experts find assumptions. **CCS**'21.

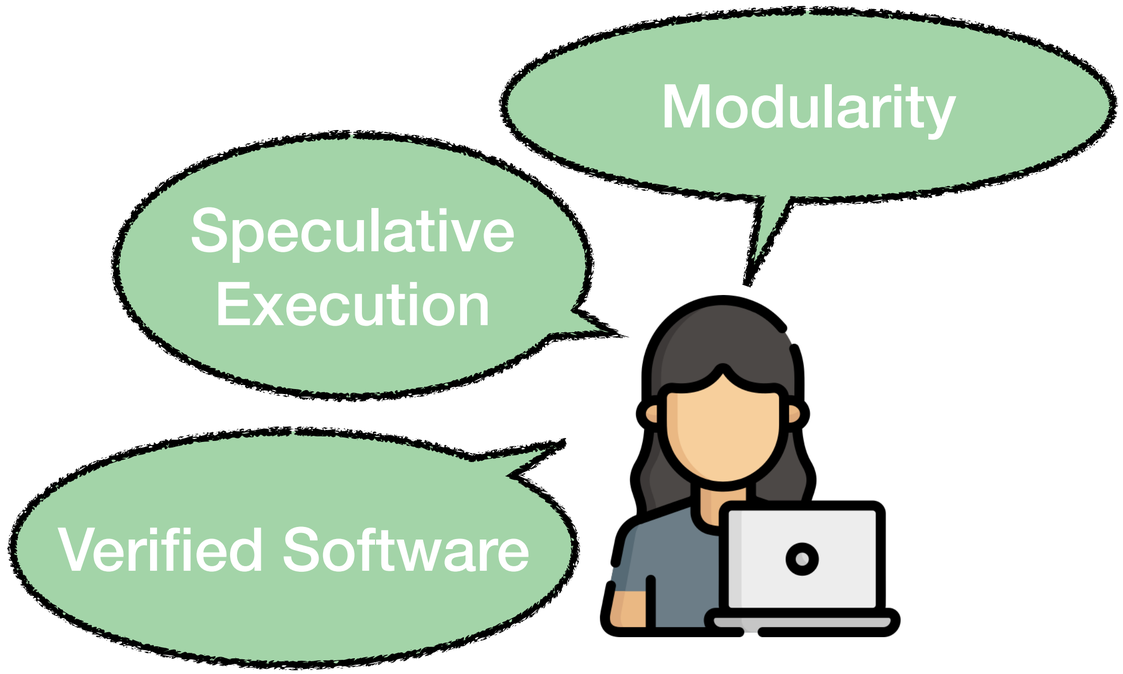
[3] LeaVe: Assumptions at Software level. **CCS**'23.
Distinguished paper.

My Journey


Verified ScarV, Sodor, DarkRiscV, Ibex
Leakage description at software level
Help for non-expert users

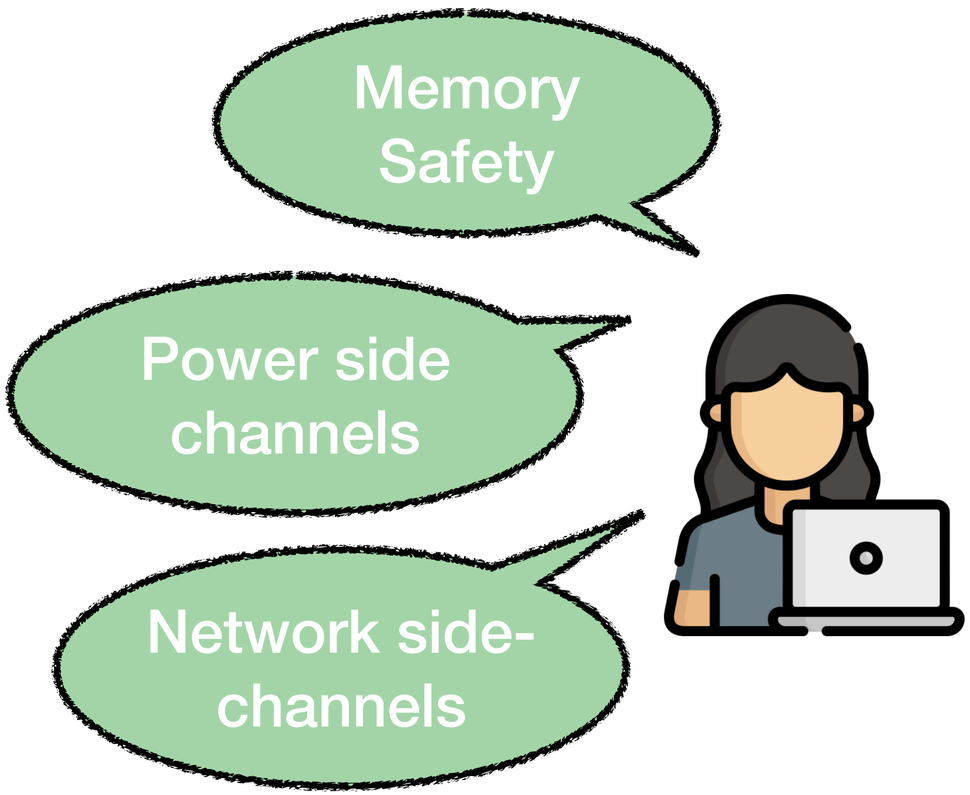

[3]
[2]
[1]

Future Work


Modularity
Speculative Execution
Verified Software


[4?]
[3]
[2]
[1]

Future Work


Memory Safety
Power side channels
Network side-channels


[4?]
[3]
[2]
[1]

Come join!