IODINE: Verifying Constant-Time Execution of Hardware

Klaus von Gleissenthall  Rami Gökhan Kıcı
Deian Stefan  Ranjit Jhala

UC San Diego
2006: Investors set out to build luxury tower
$2m per unit, built to withstand extreme winds
Two years later construction stopped
... and the tower was demolished
Why?

It was built on expandable clay

... that compresses under weight,

... causing the tower to sink
Our towers are crypto algorithms
Instead of wind, we worry about *timing* side channels.
Like reinforcing, we write constant time code

```
if (secret) x = e
```

```
x = (-secret & e) | (secret - 1) & x
```
We even verify that code really executes in constant time

HACL* is used in Firefox

Fiat Crypto is used in Chrome

miTLS influenced the TLS spec
Is this code actually safe?
It depends on what we build on!
We build on hardware!
We trust hardware to be constant time

What if it’s not?

AND       DIV
SUB       FMUL
XOR       FDIV
·         ·
We need to verify that hardware is constant time!
But How? There’s no hardware definition!

Can we use Software Methods?
Definitions Don’t Apply: Parallelism

Software
- straight line code
- sequential

Hardware
- never terminates
- parallel
- pipelined
OUTLINE

1. Definition
2. Verification: Iodine
3. Evaluation
Definition Example: FPU multiplier
MUL x y
Attacker can observe timing of outputs
Multiplies numbers \( x \) and \( y \)

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

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

always @(posedge clk) begin
    flp_res <= ... //compute \( x \times y \)
end
```
Set flag iszero, if x or y is zero

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

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

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```
If iszero is set, return zero

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

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

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```
else, compute \texttt{flp\_res} along slow path

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

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

always @(posedge clk) begin
  flp_res <= ... //compute x*y
end
```
How do we show that the multiplier is not constant time?
What’s the timing of a pipelined computation?

Stage 1
Stage 2
Stage 3
What’s the timing of a pipelined computation?

Stage 1

Stage 2

Stage 3
What’s the timing of a pipelined computation?

Stage 1

Stage 2

Stage 3

3?
But what, if instructions influence each other?
But what, if instructions influence each other?

Stage 1

Stage 2

Stage 3
But what, if instructions influence each other?

Stage 1

Stage 2

Stage 3
Instead, we...track which inputs influenced a register...at which cycle

\[
\begin{align*}
\text{in} & \at 1 \\
\text{in}_1 & \at 4 \rightarrow \text{out} \\
\text{in}_2 & \at 6
\end{align*}
\]
The set of all cycles is the influence set.

\[
\begin{align*}
\text{in} & \rightarrow \text{out} \quad \cdots \\
\text{in}_1 & \rightarrow \text{out} \\
\text{in}_2 & \rightarrow \text{out} \\
1 & \rightarrow 4 \\
4 & \rightarrow 6 \\
6 & \rightarrow 1
\end{align*}
\]

... our notion of timing
Different influence sets = not constant time
For any two executions ... 
...

... regardless of secrets
Timing of outputs must be the same
Timing of **outputs** must be the same

all **outputs**, produce same influence sets
Timing of outputs must be the same for all outputs, producing the same influence sets.
Find two executions s.t. outputs have different influence sets.
Two Executions

take *Fast Path*

take *Slow Path*
Fast Path: $x=0$ and $y=1$
assign iszero = (x==0 || y==0);

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

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

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>
assign iszero = (x==0 || y==0);

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

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

cycle  x  y  flp_res  out
0 0 1 1 ⊥ ⊥
1 0 1 ⊥ 0
assign iszero = (x==0 || y==0);

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

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

cycle | x  | y  | flp_res | out
------|----|----|--------|----
   0  | 0  | 1  |   ⊥    | ⊥  |
   1  | 0  | 1  |   ⊥    | 0  |
...
  k-1 | 0  | 1  |    0   | 0  |
assign iszero = (x==0 || y==0);

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

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

cycle  x  y  flp_res  out
0      0  1  ⊥     ⊥
1      0  1  ⊥     0
...
k-1    0  1  0     0
k      0  1  0     0
Influence Sets
assign iszero = (x==0 || y==0);

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

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
assign iszero = (x==0 || y==0);

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

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

cycle  x  y  flp_res  out
0  {0}  {0}  ∅  ∅
1  {1}  {1}  ∅  {0}
assign iszero = (x==0 || y==0);

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

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

cycle   x    y    flp_res    out
0       {0}   {0}   ∅      ∅
1       {1}   {1}   ∅      {0}
...
k-1     {k-1} {k-1} {0}   {k-2}
assign iszero = (x==0 || y==0);

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

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

cycle x y flp_res out
---
0 \{0\} \{0\} \emptyset \emptyset
1 \{1\} \{1\} \emptyset \{0\}
...
k-1 \{k-1\} \{k-1\} \{0\} \{k-2\}
k \{k\} \{k\} \{1\} \{k-1\}
Slow Path: $x=1$ and $y=1$
Slow Path: Influence Sets

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

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

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

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{}</td>
<td>{}</td>
<td></td>
<td>∅</td>
</tr>
</tbody>
</table>
assign iszero = (x==0 || y==0);

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

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

cycle  x   y   flp_res   out

0   {0}  {0}    ∅    ∅
1   {1}  {1}    ∅    {0}
Slow Path: Influence Sets

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

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

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

cycle | x | y | flp_res | out

| 0   | {0} | {0} | ∅   | ∅   |
| 1   | {1} | {1} | ∅   | {0} |
| ... |
| k-1 | {k-1} | {k-1} | {0} | {k-2} |
Slow Path: Influence Sets

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

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

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

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0}</td>
<td>{0}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
<td>{1}</td>
<td>0</td>
<td>0,1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-1</td>
<td>{k-1}</td>
<td>{k-1}</td>
<td>0</td>
<td>{0, k-2}</td>
</tr>
<tr>
<td>k</td>
<td>{k}</td>
<td>{k}</td>
<td>1</td>
<td>{0, k-1}</td>
</tr>
</tbody>
</table>
# Fast Path vs. Slow Path

<table>
<thead>
<tr>
<th>cycle</th>
<th>$x$</th>
<th>$y$</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${0}$</td>
<td>${0}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>1</td>
<td>${1}$</td>
<td>${1}$</td>
<td>$\emptyset$</td>
<td>${0}$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k-1$</td>
<td>${k-1}$</td>
<td>${k-1}$</td>
<td>${0}$</td>
<td>${k-2}$</td>
</tr>
<tr>
<td>$k$</td>
<td>${k}$</td>
<td>${k}$</td>
<td>${1}$</td>
<td>${k-1}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle</th>
<th>$x$</th>
<th>$y$</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
</table>
| 0     | $\{0\}$ | $\{0\}$ | $\emptyset$ | $\emptyset$ | ✓
| 1     | $\{1\}$ | $\{1\}$ | $\emptyset$ | $\{0\}$ | ✓
| ...   |       |       |         |      |
| $k-1$ | $\{k-1\}$ | $\{k-1\}$ | $\{0\}$ | $\{k-2\}$ | ✓
| $k$   | $\{k\}$ | $\{k\}$ | $\{1\}$ | $\{0,k-1\}$ | ✗
Different influence sets for out! Not constant time!
Outline

1. Definition
2. Verification: Iodine
3. Evaluation
Equivalent:
Liveness Equivalence
A register is \( t \)-live,

\[ x \text{ is } t \text{-live} \]

...if it’s influenced by inputs from cycle \( t \)
A register is \( t \)-live,

\[ x \text{ is } t \text{-live } \iff t \in \{1, 4, 6, t\} \]

...if it’s influenced by inputs from cycle \( t \)
A register is \( t \text{-live}, \)

\[
x \text{ is } t \text{-live} \iff t \in \mathbb{N}
\]

... i.e., if \( t \) is in its influence set
Two executions are *liveness equivalent*, if …
Two executions are *liveness equivalent*, if ...
Two executions are \textit{liveness equivalent}, if...

all outputs are \textit{t-live} equivalent, for all $t$
Constant Time \iff Liveness Equivalence.
Show that the multiplier is not constant time (using liveness)
**Fast Path: Liveness**

```verilog
class iszero = (x==0 || y==0);

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

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

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L</td>
<td>L</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>
assign iszero = (x==0 || y==0);

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

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

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L</td>
<td>L</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>L</td>
</tr>
</tbody>
</table>
assign iszero = (x==0 || y==0);

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

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

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L</td>
<td>L</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>L</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-1</td>
<td>D</td>
<td>D</td>
<td>L</td>
<td>D</td>
</tr>
</tbody>
</table>
assign iszero = (x==0 || y==0);

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

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

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L</td>
<td>L</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>L</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-1</td>
<td>D</td>
<td>D</td>
<td>L</td>
<td>D</td>
</tr>
<tr>
<td>k</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>
assign iszero = (x==0 || y==0);

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

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

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L</td>
<td>L</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>
assign iszero = (x==0 || y==0);

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

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

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L</td>
<td>L</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>L</td>
</tr>
</tbody>
</table>
Slow Path: Liveness

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

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

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

cycle | x | y | flp_res | out
------|---|---|---------|---
   0  | L | L |   D     | D
   1  | D | D |   D     | L
   ...|
   k-1| D | D |   L     | D
assign iszero = (x==0 || y==0);
always @(posedge clk) begin
  if (iszero)
    out <= 0;
  else
    out <= flp_res;
end
always @(posedge clk) begin
  flp_res <= ... //compute x*y
end
# Fast Path vs. Slow Path

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L</td>
<td>L</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>L</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-1</td>
<td>D</td>
<td>D</td>
<td>L</td>
<td>D</td>
</tr>
<tr>
<td>k</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

- ✓: Correct path
- ✗: Incorrect path
Let’s fix the FPU ...

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

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

always @(posedge clk) begin
  flp_res <= ... //compute x*y
end
```
Let’s fix the FPU ...

... by adding a constant-time mode
(like ARM DIT)
...if ct is set, always take the slow path

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

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

always @(posedge clk) begin
  flp_res <= ... //compute x*y
end
```
Verify constant time, if \( ct \) is set
For register $x$, we introduce liveness bit $x^\cdot$

```verilog
class
always @(posedge clk) begin
  if (ct)
    if (iszero)
      out <= 0;
    else
      out <= flp_res;
  end
...
assign iszero = (x==0 || y==0);
```
For register $x$, we introduce liveness bit $x^\cdot$
For register $x$, we introduce liveness bit $x^\cdot$

```verilog
assign iszero = (x==0 || y==0);
assign iszero^\cdot = (x^\cdot \lor y^\cdot);

always @(posedge clk) begin
    if (ct)
        out <= flp_res;
        out^\cdot = (flp_res^\cdot \lor ct^\cdot);
    else
        if (iszero)
            out <= 0;
            out^\cdot = (flp_res^\cdot \lor ct^\cdot \lor iszero^\cdot);
    ...
```
assign iszero = (x==0 || y==0);
assign iszero\* = (x\* \lor y\*);

always @(posedge clk) begin
  if (ct)
    begin
      out <= flp_res;
      out\* = (flp_res\* \lor ct\*);
    end
  else
    begin
      if (iszero)
        begin
          out <= 0;
          out\* = (flp_res\* \lor ct\* \lor iszero\*);
        end
    end

...
Make two copies
Make two copies

Verify $\text{out}^\bullet_L = \text{out}^\bullet_R$, for an arbitrary $t$
Overview

1. Definition
2. Verification: Iodine
3. Evaluation
Iodine: Architecture

Verilog Assumptions

Intermediate Language

Verification Conditions

... finds proofs automatically!
Iodine: Architecture

- What can Iodine verify?
- What is the *annotation burden*?
- How *efficient* is Iodine?
What can Iodine verify?

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

- What can Iodine verify?
- What is the annotation burden?
- How efficient is Iodine?
Don’t have to prove constant time unconditionally

... but can rely on assumptions
Assume that registers are public, i.e., *free of secrets*

- Instructions *are public*
- Memory access pattern *are public*
- Reset bits *are public*
EVALUATION: Annotations

- MIPS
- RISC-V
- SHA-256
- FPU
- ALU
- FPU2
- RSA

#Annotations

Flush
Public
Recent Work: 

Infer annotations, automatically
Iodine: Architecture

- Can we applicable is Iodine?
- What is the annotation burden?
- How efficient is Iodine?
How efficient is Iodine?

- MIPS
- RISC-V
- SHA-256
- FPU
- ALU
- FPU2
- RSA

Checking time in s
SUMMARY

1. Definition
2. Verification: Iodine
3. Evaluation

https://iodine.programming.systems
BACKUP
## EVALUATION: Table

<table>
<thead>
<tr>
<th>Name</th>
<th>#LOC</th>
<th>#Flush</th>
<th>#Ind</th>
<th>CT</th>
<th>Check (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS</td>
<td>434</td>
<td>31</td>
<td>2</td>
<td>✓</td>
<td>1.329</td>
</tr>
<tr>
<td>RISC-V</td>
<td>745</td>
<td>50</td>
<td>19</td>
<td>✓</td>
<td>1.787</td>
</tr>
<tr>
<td>SHA-256</td>
<td>651</td>
<td>5</td>
<td>3</td>
<td>✓</td>
<td>2.739</td>
</tr>
<tr>
<td>FPU</td>
<td>1182</td>
<td>0</td>
<td>0</td>
<td>✓</td>
<td>12.013</td>
</tr>
<tr>
<td>ALU</td>
<td>913</td>
<td>1</td>
<td>5</td>
<td>✓</td>
<td>1.595</td>
</tr>
<tr>
<td>FPU2</td>
<td>272</td>
<td>3</td>
<td>4</td>
<td>✗</td>
<td>0.705</td>
</tr>
<tr>
<td>RSA</td>
<td>870</td>
<td>4</td>
<td>0</td>
<td>✗</td>
<td>1.061</td>
</tr>
<tr>
<td>Total</td>
<td>5067</td>
<td>94</td>
<td>33</td>
<td>-</td>
<td>21.163</td>
</tr>
</tbody>
</table>
Leftovers
EVALUATION: Annotations