A Hardware Design Language for Timing-Sensitive Information-Flow Security

Danfeng Zhang, Yao Wang, G. Edward Suh and Andrew C. Myers

Cornell University

ASPLOS 2015
Information Security via Isolation

- Memory protection
- Memory space 1
- Memory space 2

Isolation vanishes at HW level
Microarchitecture Timing Attacks

key1

key2

Data cache

Timing leaks information!
Shared HW Leaks Information

• Data cache
  – AES [Osvik et al.’05, Bernstein’05, Gullasch et. al.’11]
  – RSA [Percival’05]
• Instruction cache [Acıičmez’07]
• Computation unit [Z. Wang&Lee’06]
• Memory controller [Wang&Suh’12]
• On-chip network [Wang et al.’14]

Hardware security is in demand!
How to build **efficient** HW that **provably** controls illegal information flows?
SecVerilog: a hardware design language for timing-sensitive information-flow security
## Verifiable Secure Hardware

<table>
<thead>
<tr>
<th></th>
<th>Lightweight design</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLIFT</td>
<td>✔</td>
</tr>
<tr>
<td>Tiwari et al.’09</td>
<td></td>
</tr>
<tr>
<td>Caisson</td>
<td>✗</td>
</tr>
<tr>
<td>Li et al.’11</td>
<td></td>
</tr>
<tr>
<td>Sapper</td>
<td>✗</td>
</tr>
<tr>
<td>Li et al.’14</td>
<td></td>
</tr>
<tr>
<td>SecVerilog</td>
<td>✔</td>
</tr>
</tbody>
</table>

- **GLIFT** [Tiwari et al.’09] requires part of HW written in FSM style.
- **Caisson** [Li et al.’11] and **Sapper** [Li et al.’14] do not.
- **SecVerilog** verifies HW designs as-is.

Require part of HW written in FSM style

Verify HW designs as-is
Verifiable Secure Hardware

<table>
<thead>
<tr>
<th></th>
<th>Lightweight design</th>
<th>Fine-grained resource sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLIFT [Tiwari et al.'09]</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Caisson [Li et al.'11]</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Sapper [Li et al.'14]</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>SecVerilog</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
## Verifiable Secure Hardware

<table>
<thead>
<tr>
<th></th>
<th>Lightweight design</th>
<th>Fine-grained resource sharing</th>
<th>Low verification overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GLIFT</strong></td>
<td><img src="checkmark.png" alt="Tick" /></td>
<td><img src="checkmark.png" alt="Tick" /></td>
<td><img src="cross.png" alt="X" /></td>
</tr>
<tr>
<td>[Tiwari et al.’09]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Caisson</strong></td>
<td><img src="cross.png" alt="X" /></td>
<td><img src="cross.png" alt="X" /></td>
<td><img src="checkmark.png" alt="Tick" /></td>
</tr>
<tr>
<td>[Li et al.’11]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sapper</strong></td>
<td><img src="cross.png" alt="X" /></td>
<td><img src="checkmark.png" alt="Tick" /></td>
<td><img src="cross.png" alt="X" /></td>
</tr>
<tr>
<td>[Li et al.’14]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SecVerilog</strong></td>
<td><img src="checkmark.png" alt="Tick" /></td>
<td><img src="checkmark.png" alt="Tick" /></td>
<td><img src="checkmark.png" alt="Tick" /></td>
</tr>
</tbody>
</table>
This Work

• SecVerilog
  – Lightweight language design
  – Fine-grained resource sharing
  – Low verification overhead (compile-time)

• Formally proved security guarantees

• Formally verified MIPS processor
Security Model

• Security policy lattice
  – Information has *label* describing intended conf.
  – In general, the labels form a *lattice*
  – For this talk, a simple lattice:
    \[ S : \text{secret} \quad P : \text{public} \]

• Attacker model (at label $P$ in the talk)
  – Attacker sees contents of public HW state at each clock tick (synchronous logic)
SecVerilog

- Verilog + Security labels (in braces)

A *statically partitioned* cache in SecVerilog

```verilog
reg[31:0] {P} d0[256], d1[256];
reg[31:0] {S} d2[256], d3[256];
wire[7:0] {P} index;
wire[1:0] {P} way;
wire[31:0] in;

... case (way)
  0: begin d0[index]=in; end
  1: begin d1[index]=in; end
  2: begin d2[index]=in; end
  3: begin d3[index]=in; end
endcase
...```

![Diagram of a statically partitioned cache](image.png)
SecVerilog

• Verilog + Security labels (in braces)

A *statically partitioned* cache in SecVerilog

```verilog
reg[31:0] P d0[256], d1[256];
reg[31:0] S d2[256], d3[256];
wire[7:0] P index;
wire[1:0] P way;
wire[31:0] in;

... case (way)
  0: begin d0[index]=in; end
  1: begin d1[index]=in; end
  2: begin d2[index]=in; end
  3: begin d3[index]=in; end
endcase
...```

(Inferable) Annotation for variable declarations

• General
• Little annotation burden
• Verify HW design as-is
Static labels impede resource sharing across security labels

```verilog
reg[31:0]{P} d0[256],d1[256];
reg[31:0]{S} d2[256],d3[256];
wire[7:0]{P} index;
wire[1:0]{P} way;
wire[31:0] in;

... case (way)
  0: begin d0[index]=in; end
  1: begin d1[index]=in; end
  2: begin d2[index]=in; end
  3: begin d3[index]=in; end
endcase
...```

When `way = 0` or `1`, `in` has label `P`

When `way = 2` or `3`, `in` has label `S`
An example of partitioned cache

```verilog
reg[31:0] {P}   d0[256], d1[256];
reg[31:0] {S}   d2[256], d3[256];
wire[7:0] {P}   index;
wire[1:0] {P}   way;
wire[31:0] {Par(way)}   in;
...
case (way)
  0: begin d0[index]=in; end
  1: begin d1[index]=in; end
  2: begin d2[index]=in; end
  3: begin d3[index]=in; end
endcase
...
```

Type-level function:
- `Par(0)=Par(1)=P`
- `Par(2)=Par(3)=S`

Reduces HW resources in secure designs (<1.2%)
A Sound Type System

Soundness: reject all insecure programs

A well-typed HW design \textit{provably} enforces observational determinism

P info. at each clock tick leaks no S info.

Formal definition and proof in paper
A Permissive yet Sound Type System

- Sound type systems
- Previous static methods
- More permissive

Verifies an efficient MIPS processor
Challenges

• Soundness
  – Label channels
  – Implicit declassification (details in paper)

• Permissiveness
  – Reasoning about run-time values
Label Channel

```plaintext
1 reg{P} p;
2 reg{S} s;
3 reg{LH(x)} x;
...
4 if (s) begin x = 1; end
5 if (x == 0) begin
6   p = 0;
7 end
...
```

Type-level function:

\[
\text{LH}(0) = P \quad \text{LH}(1) = S
\]

The change of label leaks information

When \( s = 0 \)

- \( s = 0 \)
- \( \text{Initial Values} \)
- \( p = 0 \)
- \( x = 0 \)
- \( p = 1 \)
- \( x = 1 \)

When \( s = 1 \)

- \( s = 1 \)
- \( \text{Initial Values} \)
- \( p = 0 \)
- \( x = 1 \)
- \( p = 1 \)
- \( x = 1 \)

\( p = s! \)

19
No-Sensitive-Upgrade [Austin&Flanagan’09]

No update to public variable in secret context

NSU rejects secure designs

\[
\begin{align*}
\text{reg}\{S\} & \text{ hit2, hit3;} \\
\text{reg}[1:0]\{\text{Par}(\text{way})\} & \text{ way;}
\end{align*}
\]

... 

\[
\begin{align*}
\text{if (hit2 || hit3)} \\
& \text{ way } \leftarrow (\text{hit2? 2:3}); \\
\text{else} & \\
& \text{ way } \leftarrow 2; \\
\end{align*}
\]

... 

(incorrectly) rejected

From a secure MIPS processor

New label of way is always $S$

Type-level function:

\[
\begin{align*}
\text{Par}(0) = \text{Par}(1) = P \\
\text{Par}(2) = \text{Par}(3) = S
\end{align*}
\]
Our Solution

No update to public variable in secret context, if the variable is not updated in all branches

```verilog
reg{S}    hit2, hit3;
reg[1:0] {Par(way)}    way;
...
if (hit2 || hit3)
    way <= (hit2? 2:3);
else
    way <= 2;
...
```

Our solution is also more permissive than flow-sensitive systems [Hunt&Sands’06, Russo&Sabelfeld’10]
Reasoning about run-time values

```verilog
reg[31:0] {P} d0[256], d1[256];
reg[31:0] {S} d2[256], d3[256];
wire[7:0] {P} index;
wire[1:0] {P} way;
wire[31:0] {Par (way)} in;

...
case (way)
  0: begin d0[index]=in; end
  1: begin d1[index]=in; end
  2: begin d2[index]=in; end
  3: begin d3[index]=in; end
endcase
...
```

Type-level function:
Par (0) = Par (1) = P
Par (2) = Par (3) = S
**Predicate Generation**

**$P(c)$ : a true predicate before $c$ executes**

```
reg[31:0] {P} d0[256], d1[256];
reg[31:0] {S} d2[256], d3[256];
wire[7:0] {P} index;
wire[1:0] {P} way;
wire[31:0] {Par (way)} in;

...  P(c): (way = 0)
```

Type-level function:

Par(0) = Par(1) = $P$

Par(2) = Par(3) = $S$

$\text{Par(way)} \subseteq P$, when (way = 0)?
Soundness

Permissiveness

Type system

Other analyses

Variables not always updated

Predicate generation
The type system only assumes correctness of other program analyses.
Agenda

• The SecVerilog language
• A permissive yet sound type system
• Evaluation
Formally Verified MIPS Processor

Rich ISA: runs OpenSSL with off-the-shelf GCC

Classic 5-stage in-order pipeline

– Typical pipelining techniques
  • data hazard detection
  • stalling
  • data bypassing/forwarding
Baseline: Unmodified and insecure

Unverified: Secure but not verified

Verified: Secure and verified

Overhead of SecVerilog
Verification Overhead

• Verification time
  2 seconds for the complete MIPS processor

• Designer effort
  – Annotation burden
    one label/variable declaration (mostly inferable)
  – Added logic due to imprecision
    27 LOC to establish necessary invariants

No added overhead on software: does not affect cycle-to-cycle behavior
## Overhead of Hardware Resources

Secure but not verified

<table>
<thead>
<tr>
<th></th>
<th>Unverified</th>
<th>Verified</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay w/ FPU (ns)</td>
<td>4.20</td>
<td>4.20</td>
<td>0%</td>
</tr>
<tr>
<td>Delay w/o FPU (ns)</td>
<td>1.67</td>
<td>1.66</td>
<td>-0.6%</td>
</tr>
<tr>
<td>Area ($\mu m^2$)</td>
<td>401420</td>
<td>402079</td>
<td>0.2%</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>575.6</td>
<td>575.6</td>
<td>0%</td>
</tr>
</tbody>
</table>

Much lower than comparable designs in previous work (4% to 660%)
Baseline (unmodified)

Overhead of secure processor

Verified

Secure and verified
# Overhead of Hardware Resources

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Verified</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay w/ FPU (ns)</td>
<td>4.20</td>
<td>4.20</td>
<td>0%</td>
</tr>
<tr>
<td>Delay w/o FPU (ns)</td>
<td>1.64</td>
<td>1.66</td>
<td>1.2%</td>
</tr>
<tr>
<td>Area ($\mu m^2$)</td>
<td>399400</td>
<td>402079</td>
<td>0.7%</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>575.5</td>
<td>575.6</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

The novel type system of SecVerilog verifies efficient design
Performance Overhead on Software

9% in average
Conclusion

• SecVerilog: a HDL with info. flow control
  – Lightweight language design
  – Fine-grained resource sharing
  – Low verification overhead (compile-time)
• A permissive yet sound type system
  – Integrating program analyses in modular way
• SecVerilog verifies efficient HW designs, with low verification overhead
Questions?