Semantic approaches to hardware vulnerabilities

Sébastien MICHELLAND
1st. year PhD, with Christophe DELEUZE and Laure GONNORD
sebastien.michelland@lcis.grenoble-inp.fr
Wait, what hardware vulnerabilities?

**Fault injections!**
Well-known attack on RSA [Bar-El et al.(2006)]:

- **Message** → **Sign** → **Good signature** $s$
- **Inject a fault!** → **Bad signature** $\hat{s}$
- $GCD(s - \hat{s}, N) = \text{Private key!}$

**Side channels!**
Power, heat, timing data revealing the program’s internal state.
These vulnerabilities break the program...

In a lot of ways, approximated by **fault models**:

- Corrupt data in registers
- Skip instructions
- Jump anywhere in the program
- Disrupt instruction decoding
- Leak instructions' execution times [Winderix et al. (2021)]

We really want to automate them away!

% clang prog.c -harden-faults=instruction-skip ...
... or, rather, they break the *semantics*.

**Assembler with instruction skip transition:**

\[
\sigma \vdash i \rightarrow \sigma' \\
\text{Initial state} \quad \text{Instruction} \quad \text{Final state}
\]

\[
\begin{align*}
\sigma \vdash i_1 \rightarrow \sigma_{\text{tmp}} & \quad \sigma_{\text{tmp}} \vdash i_2 \rightarrow \sigma' \\
\sigma \vdash (i_1; i_2) \rightarrow \sigma' & \quad \text{SEQUENCE} \\
\sigma \vdash i_2 \rightarrow \sigma' & \quad \text{SKIP!}
\end{align*}
\]

Thinking semantics leads right into relevant questions!

- How many skips can occur? How frequently?
- What if we skip the terminator of a block?
They expose architectural details.

To model side channels, add a trace of observable leaked data:

\[ \sigma \vdash i \rightarrow (\sigma', \tau) \]

Initial state \quad Instruction \quad Final state \quad Trace

Ok, but how long is \texttt{mov r1, r2} in the first place?

Problem: faults expose architectural details.

A tuned semantics, unlike assembler, can capture these!
Yet, existing work apparently doesn’t bridge this gap.

**Assembler level:**
- Friendly to reason with to design countermeasures
- Compiler can sometimes help with automation [Winderix et al. (2021)]
- But fault models are hitting an accuracy wall [Laurent et al. (2018)]

**Meanwhile, microarchitecture level:**
- Often models the entire circuit’s RTL or even lower-level
- Focuses on finding the effects of faults
Yet, existing work apparently doesn’t bridge this gap.

Assembler level:
- Friendly to reason with to design countermeasures
- Compiler can sometimes help with automation [Winderix et al.(2021)]
- But fault models are hitting an accuracy wall [Laurent et al.(2018)]

Can we capture a useful abstraction middle ground?

Meanwhile, microarchitecture level:
- Often models the entire circuit’s RTL or even lower-level
- Focuses on finding the effects of faults
Semantics helps by being formal...

Let’s protect against the instruction-latency side channel:

\[ P_1 \]

- \text{branch} \rightarrow \text{add (2 cycles)} \rightarrow \text{mov (1 cycle)}

\[ P_2 \]

- \text{branch} \rightarrow \text{nop1 (1 cycle)} \rightarrow \text{add (2 cycles)} \rightarrow \text{mov (1 cycle)} \rightarrow \text{nop2 (2 cycles)}

Theorem (\( P_2 \) is protected!)

- \( P_2 \) computes the same result as \( P_1 \) (same \( \sigma \) → same \( \sigma' \))
- \( P_2 \) leaks no input-dependent timings (\( \tau \) is the same for all \( \sigma \))
Situation: one byte of code is skipped, offsetting all opcodes.

How do we deal with corrupted instructions at assembler level?
  ▶ Honestly: we don’t.
  ▶ Protection code is corrupted too anyway.

Any serious option will have to deal with the low-level details:
  ▶ Add an integrity check in decoder?
  ▶ Use judicious opcodes so that a chosen register cannot be leaked during the 1-2 cycles until the error is detected?
... and is known to interface well with compiler optimization.

Options for hardening at a high-level:

1. Harden after compiling [Winderix et al.(2021)];
2. Avoid compiler interference with −O0;
3. Encode the counter-measure’s properties in C/IR/ASM semantics to force the compiler to preserve them [Vu et al.(2020)].

Option #3 brings us right in sight of the altered semantics we’ve been studying!
Conclusion

- Semantics functions like a flexible abstraction level.
- We can use it to capture important architectural aspects and involve hardware in countermeasure design!
- We can also use it as a bridge to compiler automation and compiler optimization!
Semantics functions like a flexible abstraction level.

We can use it to capture important architectural aspects and involve hardware in countermeasure design!

We can also use it as a bridge to compiler automation and compiler optimization!

Audience questions?
References I


References II

https://doi.org/10.1145/3377555.3377897

https://doi.org/10.1109/EuroSP51992.2021.00050