Information-Flow Analysis for a Custom Processor

Jade Philipoom

PopSi 2026

Motivation

Cryptographic bignum/vector coprocessor (Pavona's ACC/OpenTitan's OTBN)

- No compiler, little tooling
- Simple design: few instructions, predictable timing

Motivation

Cryptographic bignum/vector coprocessor (Pavona's ACC/OpenTitan's OTBN)

- No compiler, little tooling
- Simple design: few instructions, predictable timing

Uses for information flow:

- constant-time checking
- clearing sensitive values
- clobbered registers
- debugging

Design

Starting principles:

- Analyze at compile time
- Return single graph with all possible information flow
- "Sound but not complete" philosophy
- Take advantage of existing tooling and restricted context

Design

Starting principles:

- Analyze at compile time
- Return single graph with all possible information flow
- "Sound but not complete" philosophy
- Take advantage of existing tooling and restricted context

Steps:

1. Build control-flow graph (skip for this talk)
2. Encode information flow per-instruction
3. Build information-flow graph
4. Refine approach

Encoding information flow

```
- mnemonic: bn.add
  operands:
    - name: wrd
      doc: Name of the destination WDR
    - name: wrs1
      doc: Name of the first source WDR
    - name: wrs2
      doc: Name of the second source WDR
    - name: flag_group
      doc: Flag group to use. Defaults to 0.
  iflow:
    - to: [wrd, flags-all]
      from: [wrs1, wrs2]
```

Encoding information flow

In some cases, immediate arguments change information flow.

```
- mnemonic: bn.sel
  iflow:
    - to: [wrd]
      from: [wrs1, wrs2]
    - test:
        - flag == 0
      to: [wrd]
      from: [flags-c]
    - test:
        - flag == 1
      to: [wrd]
      from: [flags-m]
```

Analyzing information flow

Now that we know how information flows through each *instruction*, how does it flow through complex programs?

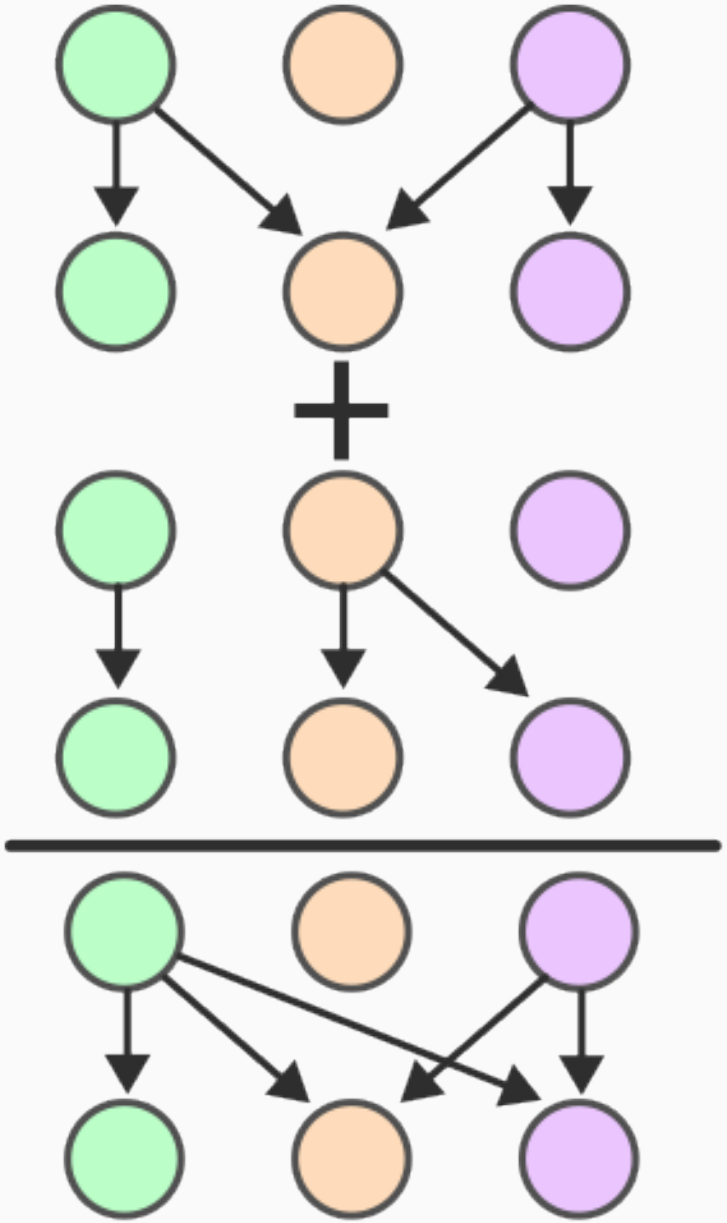
Analyzing information flow

Now that we know how information flows through each *instruction*, how does it flow through complex programs?

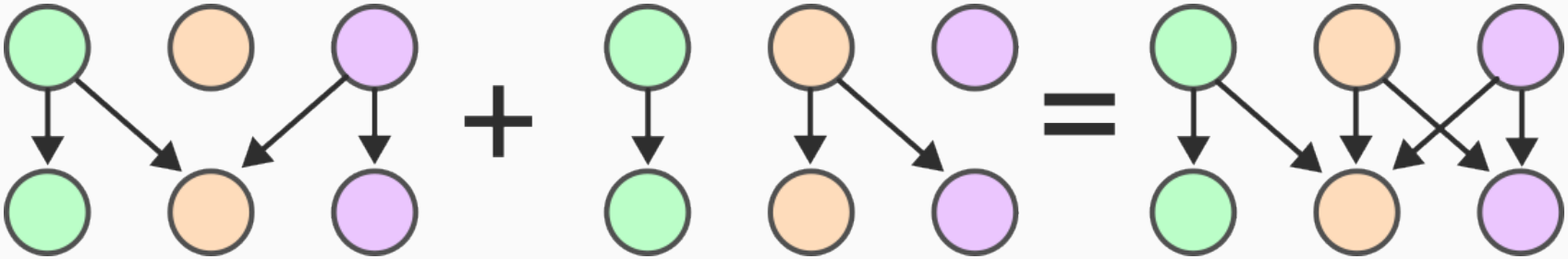
Two ways to compose information-flow graphs

- "vertically" (for instructions in sequence)
- "horizontally" (for branches or rule combinations)

Analyzing information flow



Analyzing information flow



Analyzing information flow

```
/* Returns max(w10, w11, w12) in w10. */
max3:
  bn.cmp w10, w11
  bn.sel w10, w11, w10, C
  bn.cmp w10, w12
  bn.sel w10, w12, w10, C
  ret
```

```
$ analyze_information_flow max.elf --subroutine max3
Information flow for paths ending in return:
  fg0-c <- w10,w11,w12
  fg0-l <- w10,w11,w12
  fg0-m <- w10,w11,w12
  fg0-z <- w10,w11,w12
  w10   <- w10,w11,w12
```

Refining the approach

Initial approach did not work when:

- secrets are written to memory, then non-secrets are read out and used in control flow
- number of traversals of a cycle in the control-flow graph is data-dependent

Refining the approach

Initial approach did not work when:

- secrets are written to memory, then non-secrets are read out and used in control flow
- number of traversals of a cycle in the control-flow graph is data-dependent

Solutions:

- lightweight constant propagation
- memory tracking toggle
- cycle handling by vertical *and horizontal* composition until stable

These adjustments were sufficient to cover the vast majority of the code we had. Now runs in CI!

Questions?