A New Take on Information Flow Tracking

Calvin Deutschbein
and Owyn Wyatt

Repository Access

https://tinyurl.com/rtl-trace-repo

https://github.com/horseplay333/RTL-Traces

Introduction

- Inspired by the discovery of transient execution CPU vulnerabilities
  - Spectre and Meltdown disclosure of sensitive information at the microarchitecture level
- Information flow tracking allotted each register and wire with a “shadow register”
  - Every shadow register that gets set to a non-zero value could be containing sensitive information
- To our knowledge little work to capture the whole path
  - IFT does not distinguish between immediate and intermediate sources

Problem Statement

Given a **trace of execution** from a hardware design that is instrumented with information flow tracking, **determine the path** of information flow from a given source signal to a given sink signal.

VCDs

Methodology

- We assume a design instrumented with IFT
- Map over each signal in the design
- Fix each signal as the source signal
  - Note each source's destination
  - Extract the `time_of_flow`
- Put the source/destination pair into a single dataframe
- Query to see if the destination signal is a source at a later time to identify information paths

Constructing a Graph

- Simulated and derived trace data for every signal in the design
  - Computationally costly but also more useful
- With the times-of-flow given within the trace data a graph can be constructed with the times as graph edges.
- In our design we define the `time_of_flow` as a Verilog time delta at which the tracking signal of the sink signal is set to a nonzero value.
- There are certain registers in every graph that do not flow to any other register

4 Register Design

To think about this: consider this simple four register design:

1) `key` - A register we think of as holding a sensitive information
2) `one` - An internal register in a module.
3) `two` - Another internal register in a module.
4) `out` - An output potentially visible to unprivileged user.

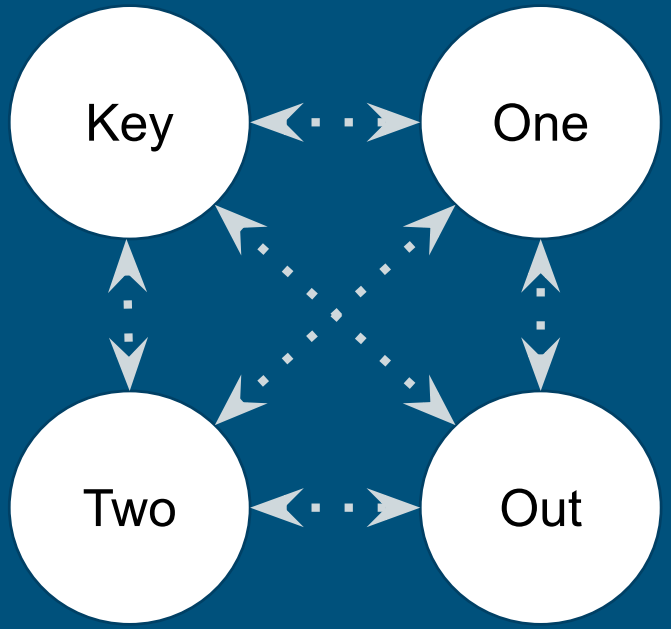
We seek to point out `one` and `two` as intermediate registers in a larger path

Example for Graph Walk Through

root	sink	Δt
key	one	1
key	two	4
key	out	7
two	out	7

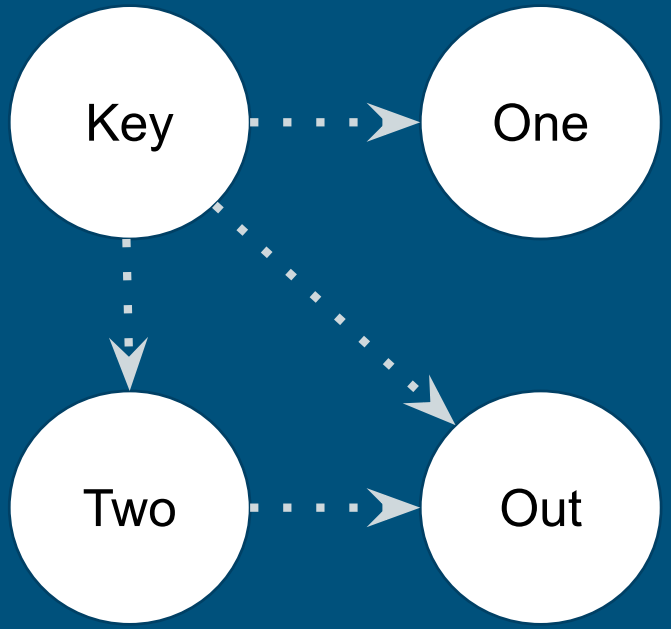
Candidate Edges

The dashed lines denote possible edges that have not been falsified



Known Non-Edges

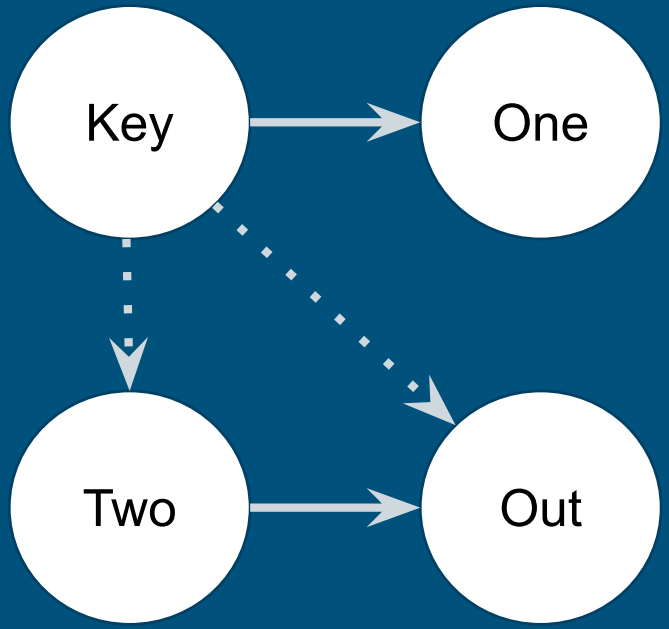
Considering the times-of-flow we note that nothing flows out of `one` or into `key`. Therefore we update our graph to look like this:



Known Edges

We consider the vertices that correspond to non-zero `time_of_flow`. A flow must be present in each root/sink pair with the smallest `time_of_flow` among that root.

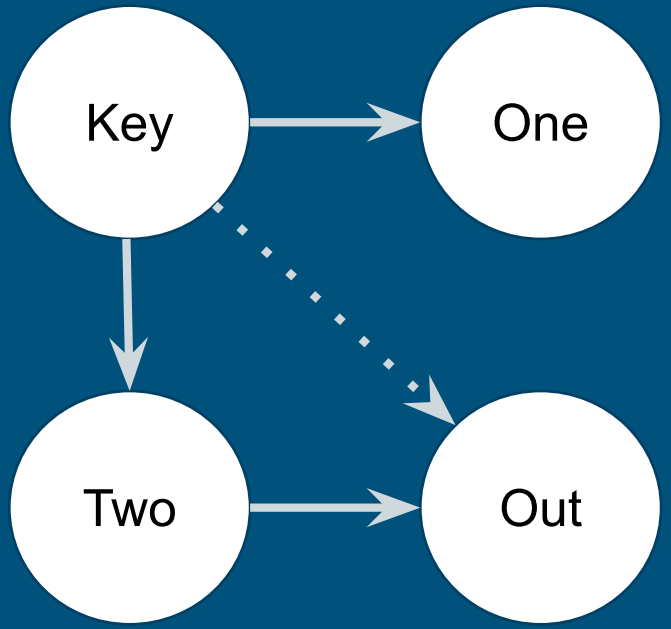
index	root	sink	Δt
0	key	one	1
1	key	two	4
2	key	out	7
3	two	out	7



Candidate Promotion

We look for a case where a source signal (`two`) can also be a sink signal with `key` as the source. This is done by looking for a time-of-flow where `two` serves as a sink signal before it serves as a source.

index	root	sink	Δt
0	key	one	1
1	key	two	4
2	key	out	7
3	two	out	7

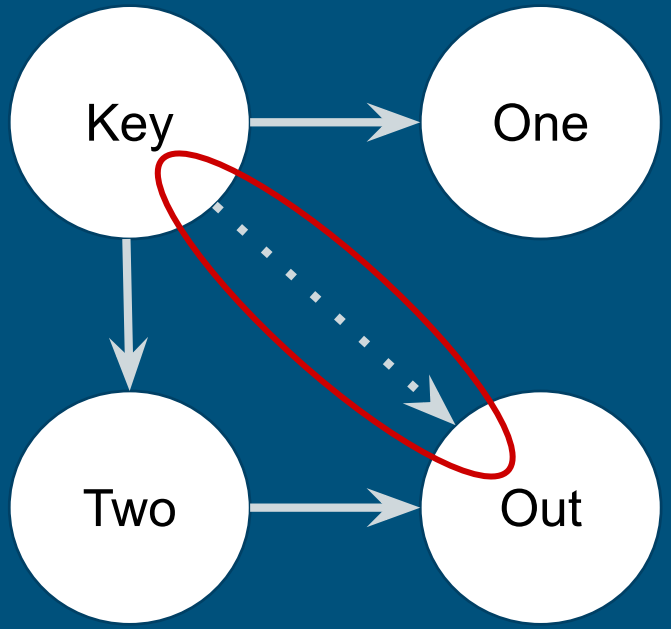


Known Limitation

In our example we have one remaining candidate edge that has not been removed. We did not find a way in our research to definitively conclude that there is no flow from `key` to `out`.

In the future we imagine a design that has an assignment such as:

```
out <= two + key
```



Case Study.

- Applied technique to PicoRV32 RISC-V CPU
- Of 232 registers in PicoRV32 182 of them are deemed non trivial
- Generated traces for all nontrivial registers
- Each register took ~3 minutes to explore
- Store the full graph as an adjacency matrix and the paths as trees rather than individuals

Results of Case Study:

The following paths and corresponding lengths were found:

Length	2	3	4	5
Count	151	130	71	59

Path of Note

```
do_waitirq ==>
mem_do_rinst ==>
mem_valid ==>
next_insn_opcode ==>
dbg_insn_opcode
```

CWE-1244: Internal Asset Exposed to Unsafe Debug Access Level or State

Case Study - Confused Deputy

- Confused Deputy
  - CWE 411: Unintended Proxy or Intermediary('Confused Deputy')
- Prior work found evidence but could not determine the presence of the confused deputy
- Access control policy:

$$AC_1 \text{ of } ACW_1: \quad R = \{P_1, P_2\}, \; W = \{P_1\}$$

$$AC_2 \text{ of } ACW_2: \quad R = \{P_3\}, \; W = \{P_2, P_3\}$$

- Atomic path between ACW_2 and P_1
  - Prior work did not have the power to do this, our new tool does

Scaling

- Trace generation stage is by far the most expensive
- PicoRV32 CPU took ~8.5 hours to generate traces
- Path construction can evaluate PicoRV32 in less then 5 minutes
- Each trace takes ~100 seconds to create
  - Write-to-disk taking most of the time
- 100x speed up for small designs
  - Useful for large designs if it is practical to create enough instances
- Goal to modify trace framework to stream events (rather then write to disk)
  - Perform validation during simulation stage

Thank you!

Any Questions?

Demo!

Trace Dependence

- Traces can impact the usefulness of discovered traces if test coverage is poor
  - Inherent weakness of trace-based technology
- Our technique restricted to information flows that actually occur
  - No theoretical flows
  - Provides useful refinement for other techniques
- Prior work uses this to view analysis as not restricted to just hardware but also operating systems

More on Candidate Edges

- Candidate edges comes from a graph algorithm perspective
- Can be more easily understood as the set of signals which can be joined by an inclusive or such as:

```
two <= key - one;
```

```
two_t <= key_t | one_t | two_t;
```

- Information flows from multiple signals simultaneously
  - Candidate edges will necessarily exist
- Understanding and exploring these edges further could be a good opportunity for collaboration/future work and/or a different tool