Aphrodite

Security Properties of RISC-V

Juni L DeYoung
NWSA-AAASPD 2023 Meeting
Western Washington University
22 March 2023

Table of contents

01 Overview

1.1
Goals

What are we doing and why are we doing it?

What are security-relevant properties of computer hardware?

Research Process

Collect

1. Model processor in software
2. Record register transfers

Analyze

3. Mine traces for properties
4. Check properties against common weaknesses

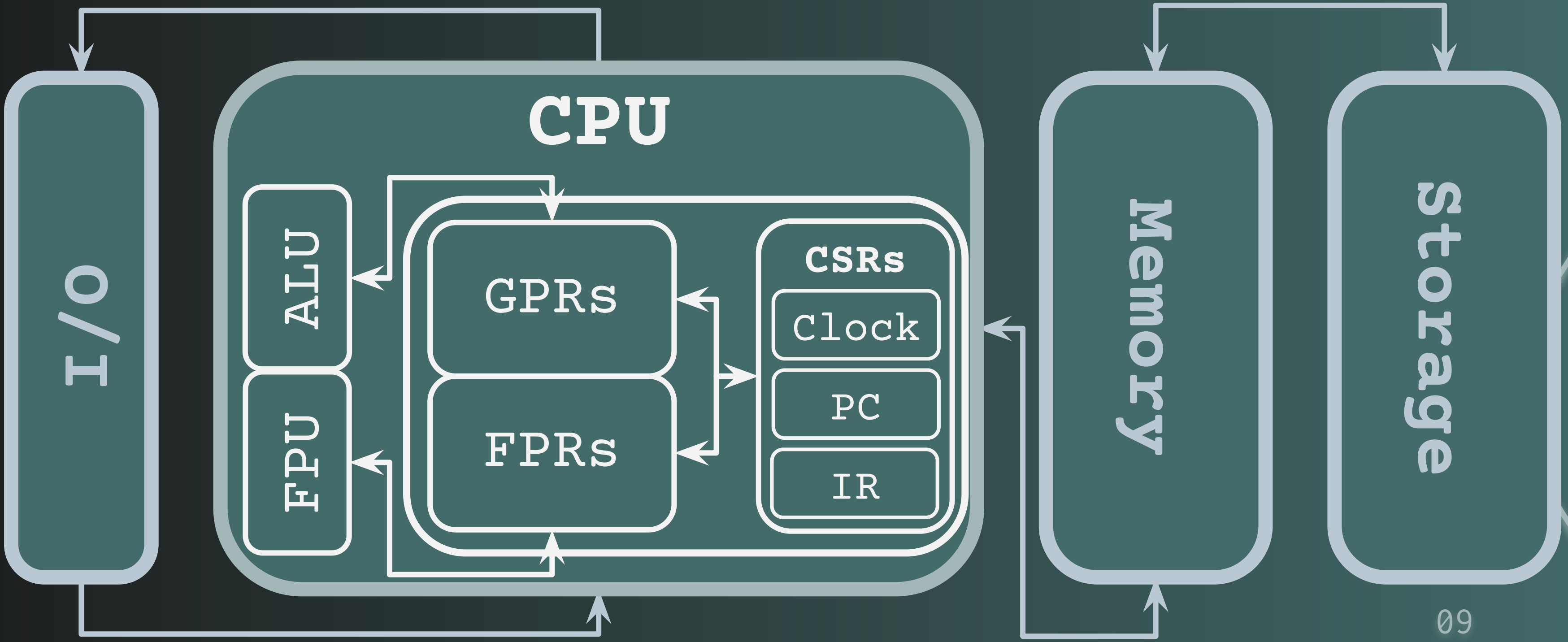
Report

5. Security properties found!

1.2 Background

What exactly are we studying here?

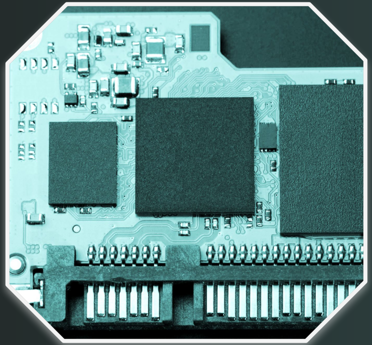
Computer Anatomy



Virtualizing Hardware

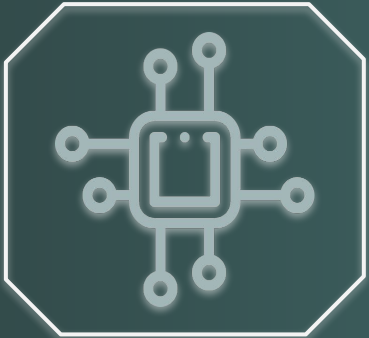
Simulation

- Recreates a processor at register transfer level (RTL)
    - Modeling the actual configuration of wires and transistors in software



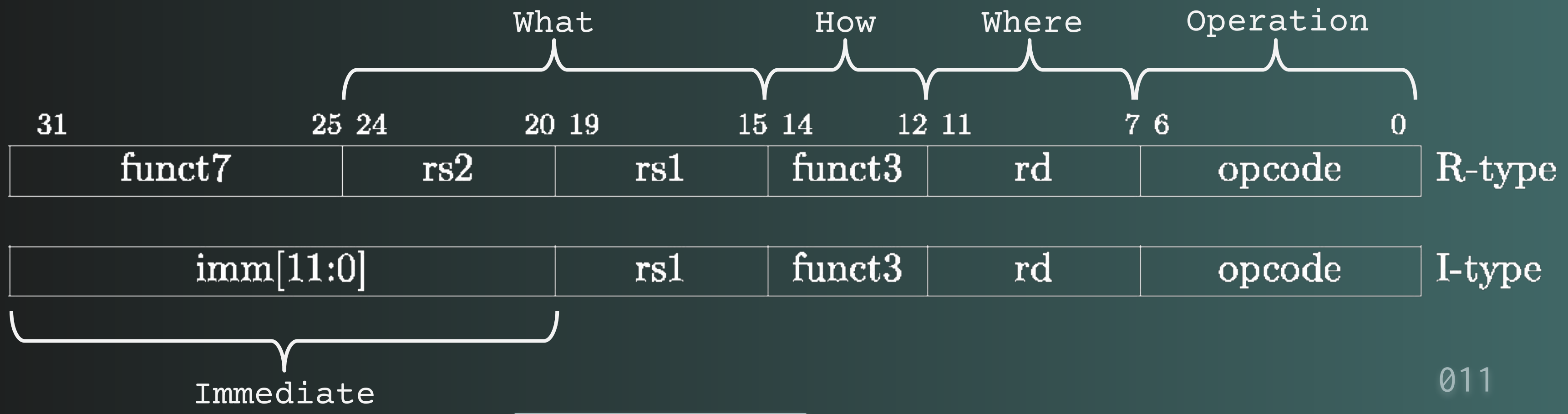
Emulation

- Recreates an instruction-set architecture (ISA)
    - Doesn’t replicate specific hardware idiosyncrasies, only its instruction set



Instructions

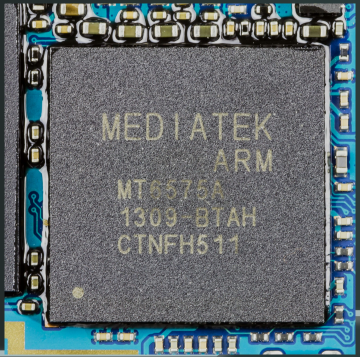
- Contained in memory
  - Addresses correspond to values in the program counter
- Control information flow through the processor
  - Performing operations (arithmetic, load/store, navigation)



ISA Paradigms

RISC

- One operation per instruction
- “Load-Store” architecture
- More difficult to write programs in assembly
- ARM



CISC

- “Microcoding”
- Instructions execute multiple operations at once
- Smaller programs
- Fewer main memory accesses
- x86

Why Study RISC?

- CISC processors are proprietary trade secrets
- RISC architectures are easier to study
  - Fixed-length instructions
  - One instruction -> one operation
- RISC-V is an open-source design
  - Funded by Intel and AMD

02

RISC-V

Emulation is the highest form of flattery

The RISC-V Spec

- Highly customizable to different configurations
- Designed for academic study **and** hardware implementation
- 32- and 64-bit variants

General Purpose Registers x0-x31

- x0 is fixed to value 0
- x1-x31 are read as booleans or (un)signed 2's complement integers

Floating-point registers f0-f31

- Correspond to IEEE standard for floating-point

Control and Status Registers

- 4096 CSRs, mostly used by the privileged architecture
  - Some use in unprivileged code, mostly as counters and timers
  - Exceptions, interrupts, traps, control transfer

Configuring Qemu

1. Download Qemu
2. Build RISC-V emulator
   a. `$ sudo apt install qemu-system-misc`

      This includes the `qemu-system-riscv64` and `riscv32` commands, which allows Qemu to boot executable files with the RISC-V `virt` emulator. It also includes several additional emulators.

The RISC-V Toolchain

In:

```
$ git clone https://github.com/riscv/riscv-gnu-toolchain --recursive
$ sudo apt-get install autoconf automake autotools-dev curl python3 [...]
$ ./configure --prefix=/opt/riscv --enable-multilib
$ sudo make linux
```

[A few hours pass]

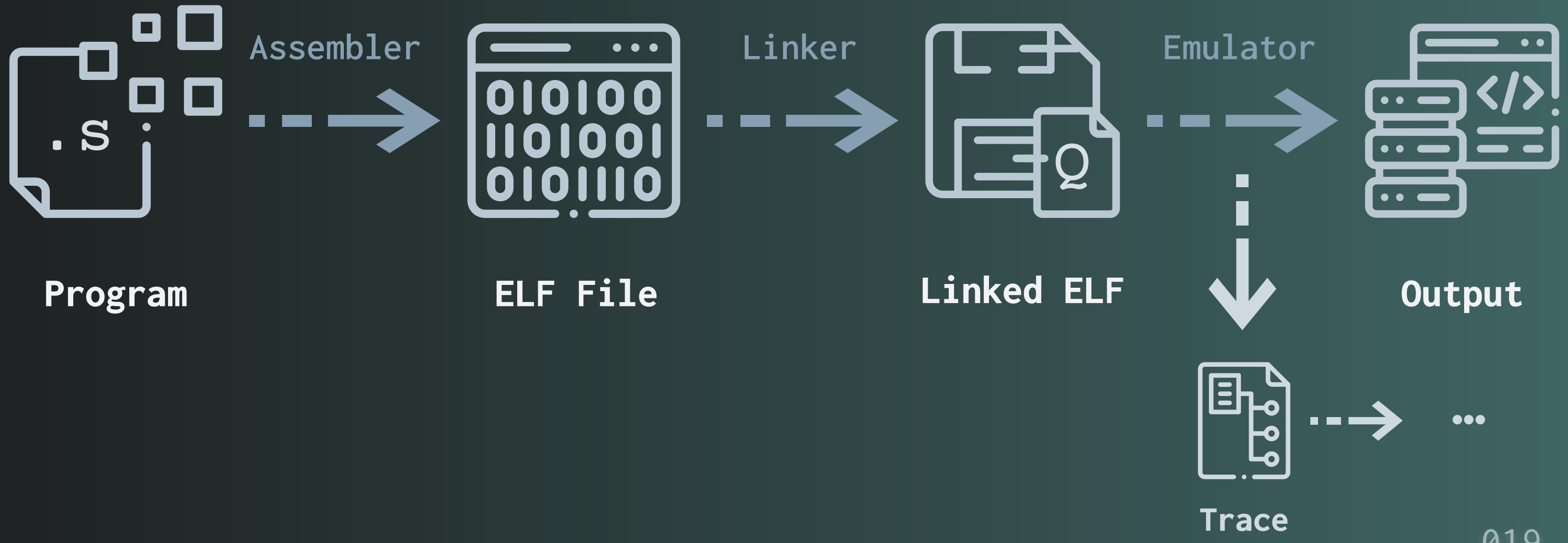
Out:

```
[...]
gcc: error: unrecognized argument in option ‘-mcmodel=medany’
gcc: note: valid arguments to ‘-mcmodel=’ are: 32 kernel large medium small
make: *** [Makefile:319: file.o] Error 1
```

“Hello World”

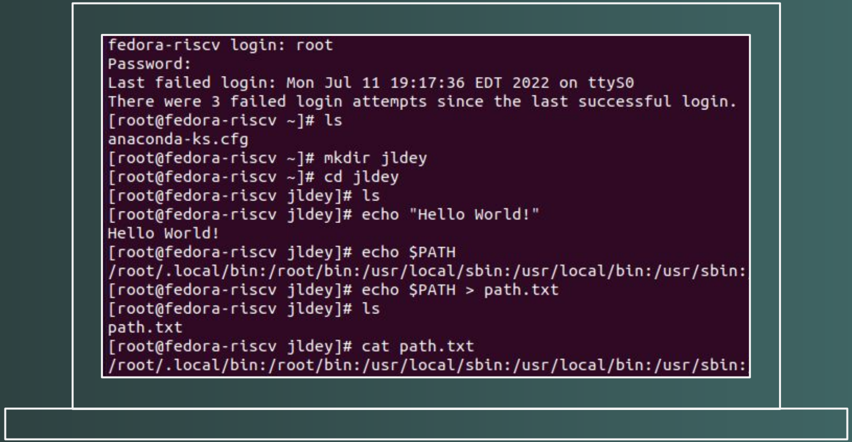
```
    .global _start          Initialize the program at “_start” label

_start:

    lui t0, 0x10000         Load address of serial port into register t0

    andi t1, t1, 0          Zero out t1
    addi t1, t1, 72         Add (int)’H’ = 72 to t1
    sw t1, 0(t0)            Send value of t1 == ‘H’ to location addressed by t0 (UART0)

    [...]                   The previous three lines are repeated for ‘e’,‘l’,’l’,’o’
                            and finally LF (line feed, aka ‘\n’)
finish:
    beq t1, t1, finish      Jump to label finish if t1==t1
```

Bare-Metal Programs on RISC-V



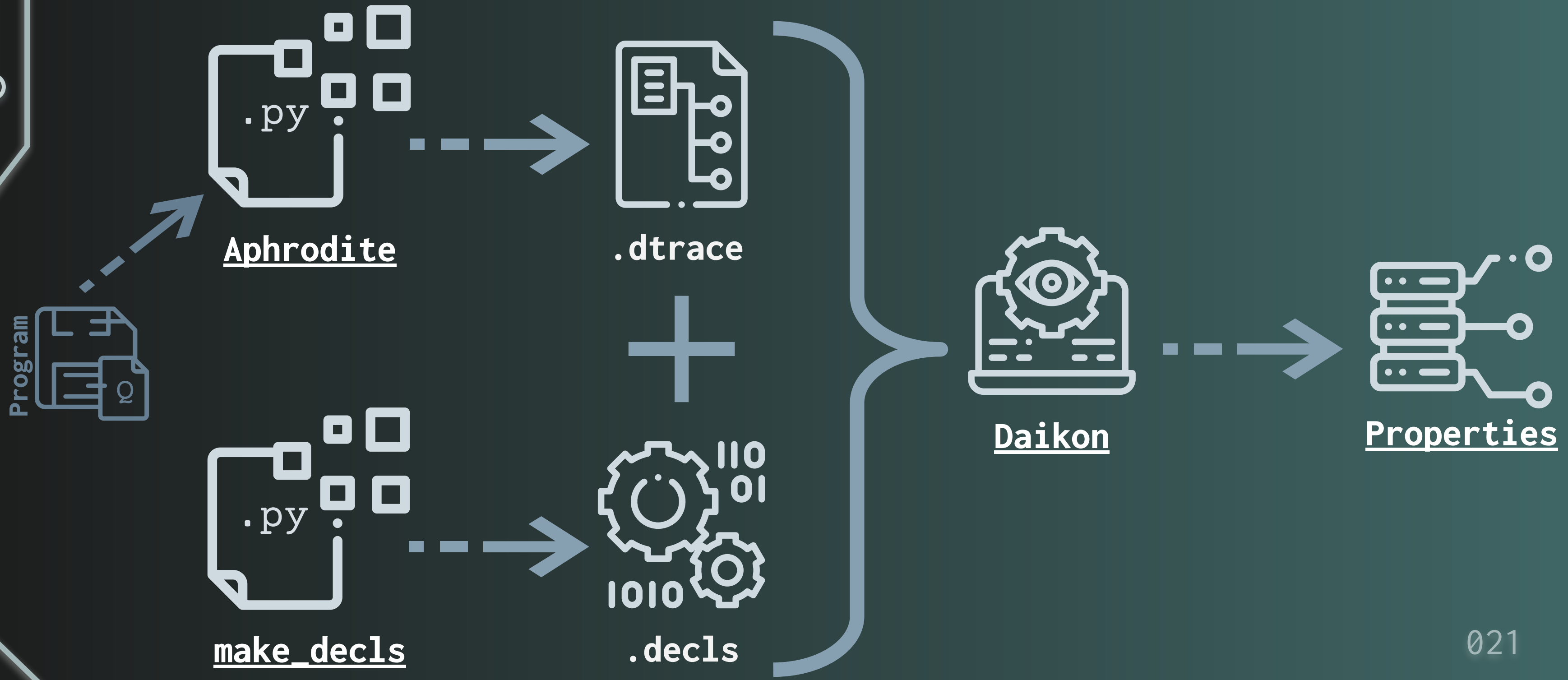
Booting Fedora

After downloading the Fedora prebuilt images, decompress and boot according to the Qemu documentation.



Above: a sample session in the Fedora emulation

Data Mining


Program
.py
Aphrodite
.dtrace
.py
make_decls
.decls
Daikon
Properties

03

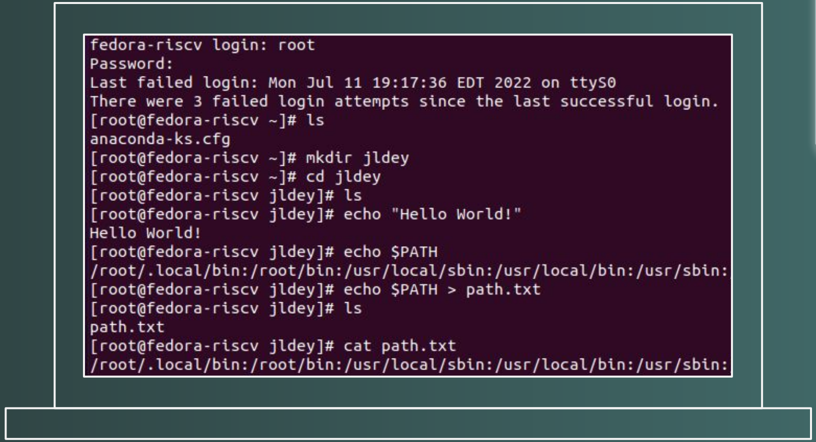
Aphrodite

Now how do we *do* all that?

Aphrodite.py

Using QEMU in Aphrodite

```
args = [
     "-machine", "virt", "-kernel", exe,
     "-monitor", "stdio", "-S",
     # options for running Fedora
     "-smp","4", "-m","2G", "-bios",
     "none",[...]
]

qemu = px.spawn("qemu-system-riscv64",
                  args, encoding="utf-8")
qemu.expect(".*(qemu)")
qemu.sendline("info registers")
qemu.expect("(qemu)")
qemu.sendline("c")
```



Above: a sample session in the Fedora emulation

Gathering Register Values

The Trick: collecting register values without changing any of them.

Tools:

- riscv-probe (CSRs)
- Qemu debugging tools (GPRs)
  - Logging (-d cpu)
  - ~~Qemu built-in “trace”~~
  - **Monitor -> info registers**
  - ~~GDB (GNU debugger)~~
    - ~~Multiarch~~
    - ~~riscv64 gdb~~
  - ~~Qemu source (fprintf hacking)~~
  - **Qemu wrapper to inject commands to monitor and write output to file**
    - ~~subprocess library~~
    - **pexpect library**

qscript (pseudocode)

```
1. Start QEMU with a linked ELF as input
    - start the VM paused (`-S`)
    - `-monitor stdio` so program can write commands to monitor
2. Ping monitor every so often (specify as commandline option?)
    - build a simple character driver to use instead of `stdio`?
    - write this output to a trace file
      - QEMU “single-step” mode (take the first N cycles)
3. Terminate VM
    - send `quit` command (or simply `q`) to monitor
       - quit condition?
              - timeout
                     -> fixed time?
                     -> based on last output change (pc?)
              - user-specified?
                     -> if the program is reading/writing to the
                          monitor console, does that mean the user can issue a `quit`?
                     -> does the user ping the script, or the monitor?
```

Trace formats

qtrace

```
i\x1b[K\x1b[Din\x1b[K\[...]
pc        0000000000001000\r
mhartid  0000000000000000\r
[...]
x0/zero 0000000000000000
x1/ra 0000000000000000
x2/sp 0000000000000000
x3/gp 0000000000000000\r
[...]
f28/ft8 0000000000000000
f29/ft9 0000000000000000
f30/ft10 0000000000000000
f31/ft11 0000000000000000\r
[...]
```

.dtrace

```
..tick():::ENTER
this_invocation_nonce
1
pc
4096
1
mhartid
0
1
[...]
f31/ft11
0
1
```

Parsing qtrace to dtrace

1. Parse register values into a list
    a. QEMU logs don't parse each timestep neatly (is this a reason not to use them?))
    b. Monitor output (qtraces) can parse each timestep
        i. Is there a potential for duplicate data?
        ii. I can parse to Daikon format at runtime and only write to file once
        iii. qtraces contain FPR values.
2. Add list generated in (1.) to a 2D list of all timesteps
    a. Get rid of any empty sublists (or completely ignore identical data)
3. Parse this 2D list into Daikon .dtrace format and write to file

Conclusion: pexpect monitor traces are a better solution that QEMU native debugging.

Parsing qtrace to dtrace

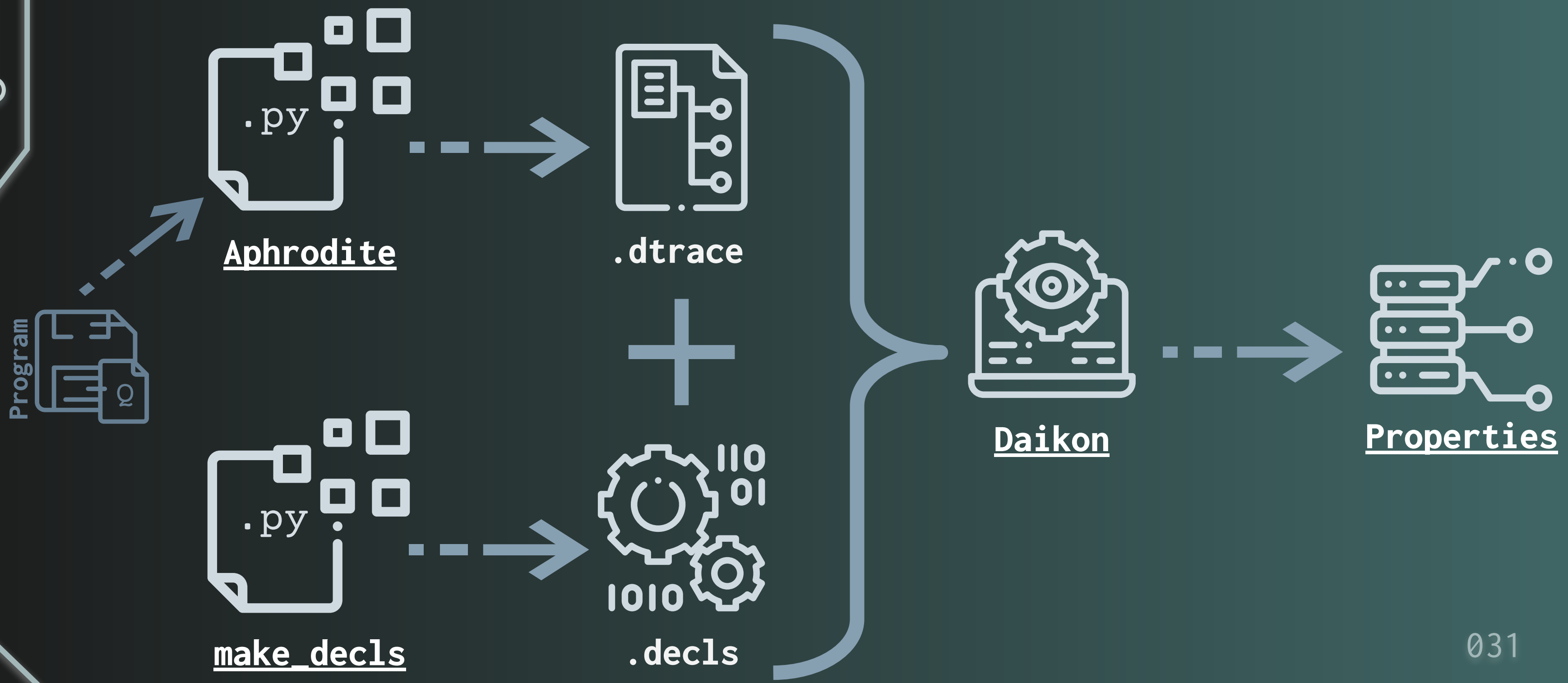
1. Create a .dtrace file and give it a unique name based on current system time
2. Spawn QEMU with initial parameters
3. While not timed out:
   a. Parse `info registers` output for register values, adding to list `vals`
   b. If `vals` is not equal to the last timepoint <u>and</u> is nonempty:
      i. Split `vals` entries into tuples: `(label, value)`
      ii. Cast the `value` hex string to an integer
      iii. Write these label/value pairs to .dtrace in the appropriate format
   c. Send next `info registers` command to QEMU
4. Quit QEMU and close .dtrace

Parsing qtrace to dtrace

```
# find all register name/value pairs on current line
# returns empty list if no register values found,
# i.e. the output was not a string of register/value pairs
vals = re.findall(r"[a-z0-9/]+\s+[0-9a-f]{16}|\w+\s+[0-9a-f]x[0-9a-f]",out)
```

```
# Parse register/value pairs into lists
for reg in vals:
    reg_val = re.split("\s+",reg)
    # hex string to int: `int("ff",16)` -> 255
    reg_val[1] = int(reg_val[1],16)
    # register name\n value \n constant 1
    dt.write(reg_val[0]+"\n"+str(reg_val[1])+"\n1\n")
    # for copying these values into the tick exit
    tpoint.append(reg_val)
```

Data Mining


Program
.py
Aphrodite
.dtrace
.py
make_decls
.decls
Daikon
Properties

Properties

```
f21/fs5 == f26/fs10
pc != 0
mhartid == 0
mip >= 0
mideleg one of { 0, 546 }
medeleg one of { 0, 45321 }
mtvec one of { 0, 2147484904L }
x0/zero == 0
f0/ft0 >= 0
[...]
f16/fa6 >= 0
f19/fs3 one of { 0, 4607182418800017408L }
f20/fs4 one of { -4616189618054758400L, 0 }
f21/fs5 one of { 0, 4472406533629990549L }
f22/fs6 >= 0
```

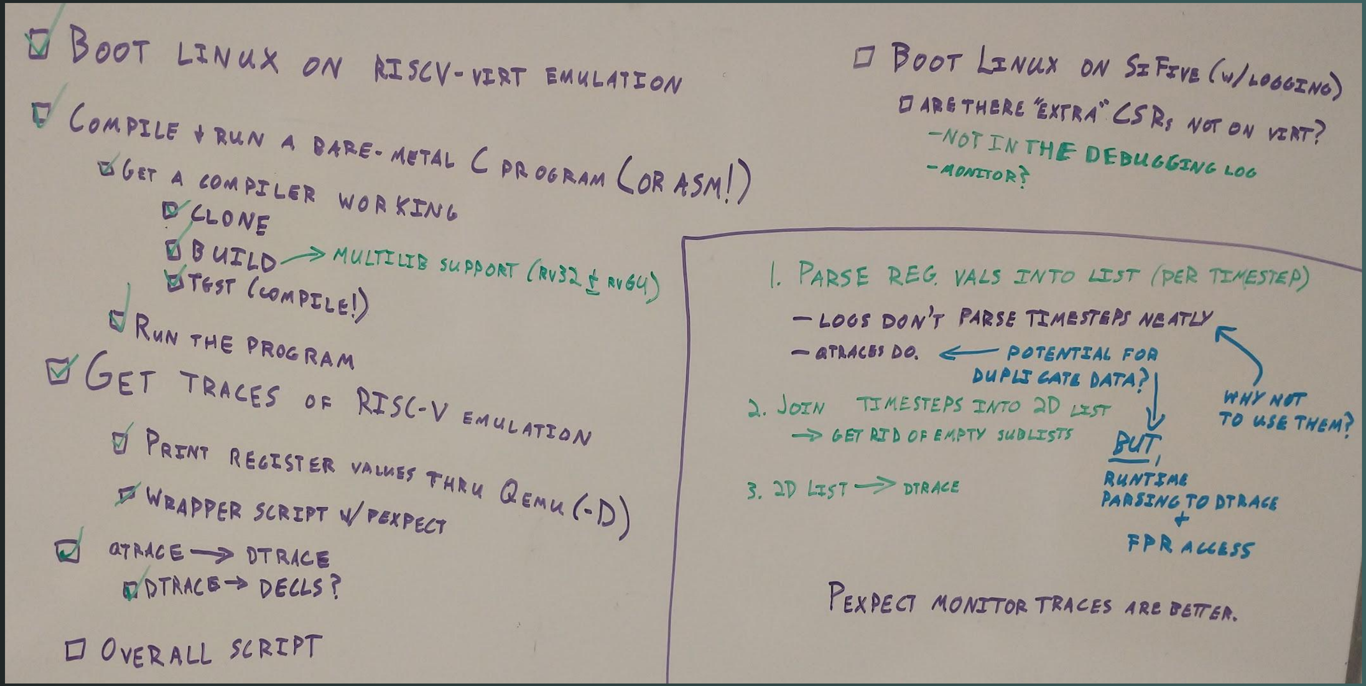
```
f23/fs7 >= 0
f24/fs8 one of { 0, 4607182418800017408L }
f25/fs9 one of { -4616189618054758400L, 0 }
[...]
pc != mhartid
[...]
mhartid <= mip
[...]
mip <= mie
[...]
mie <= mtvec
mideleg <= medeleg
[...]
mtvec >= mcause
f0/ft0 >= f20/fs4
[...]
```

Aphrodite verifies properties guaranteed by the ISA specification.

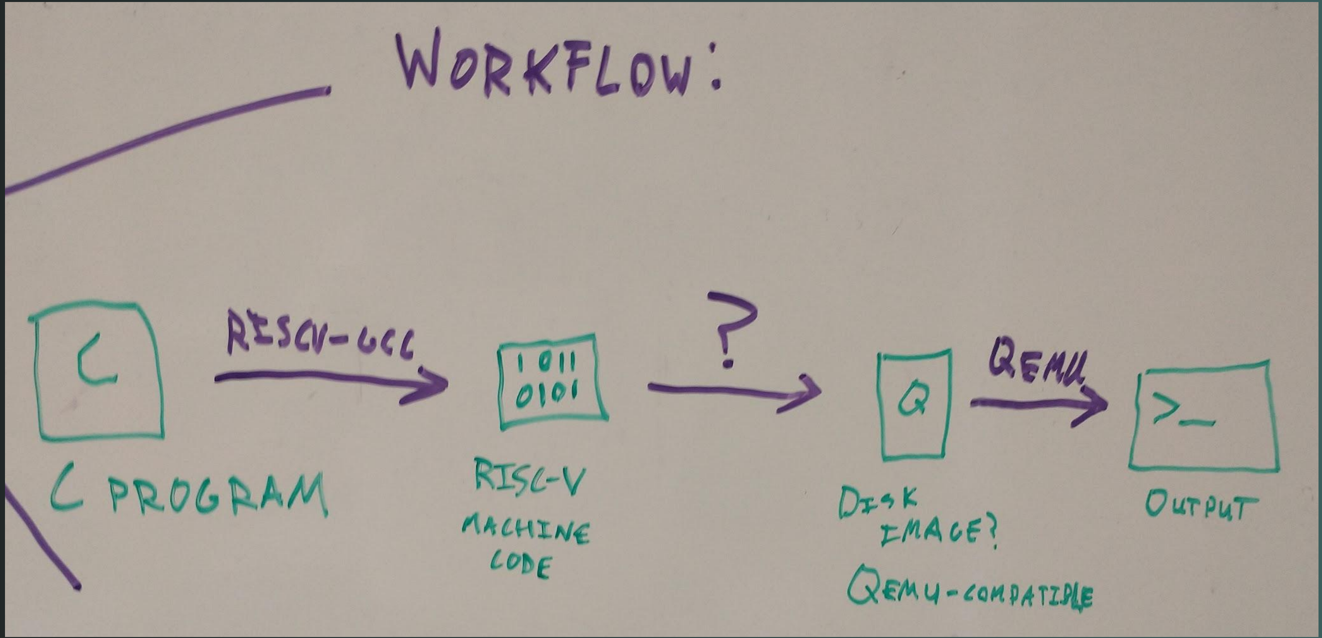
04 The REU Experience

Faff around. Find out. Get paid.

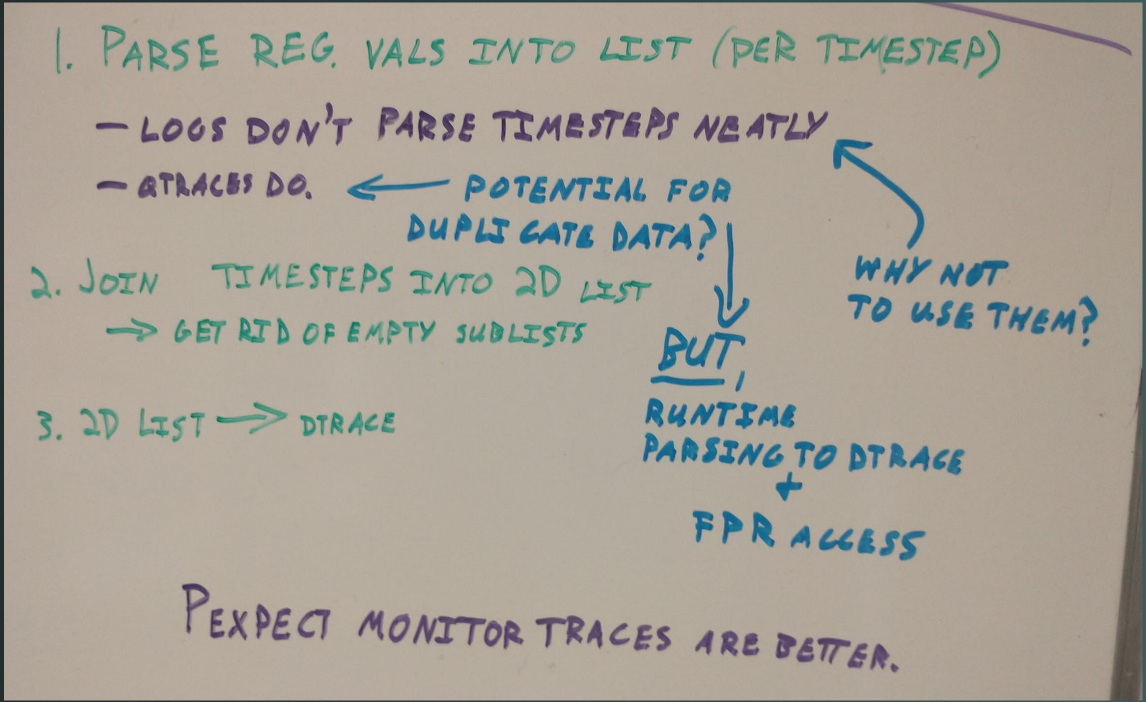
Whiteboard Notes (overall checklist)



Flowchart Draft (slide 18)



Whiteboard Notes (slide 25)



Questions?

jldeyoung@willamette.edu
jldeyoung.github.io
github.com/wu-jldeyoung