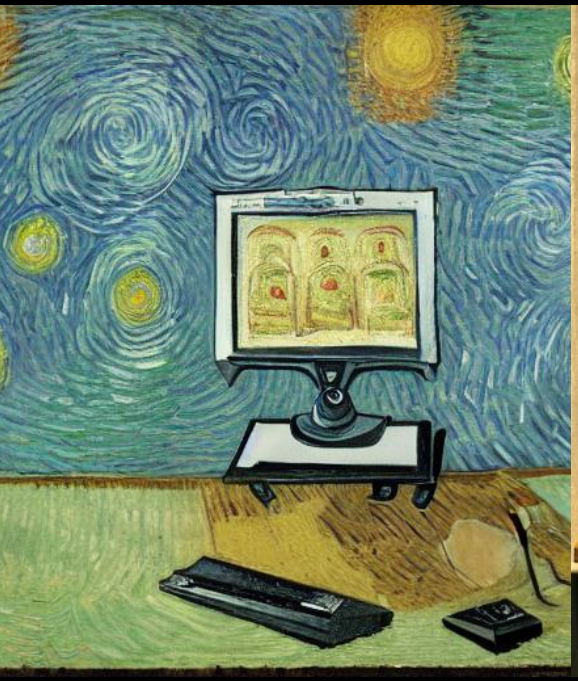
riscure

driving your security forward

# PRE-SILICON FAULT SIMULATION: HARD AND IMPORTANT

JASPER VAN WOUDENBERG
@JZVW
FDTC 2022

Automation, by Van Gogh



Shift-left, by Vermeer

Fixing vulns, by Rembrandt

# WHAT AM I TRYING TO ACHIEVE TODAY

- Show (demoable) case study about how pre-silicon FI can help
- Sprinkle in open research questions
- Lure you into research collaborations

# PRE-SILICON FAULT INJECTION

public

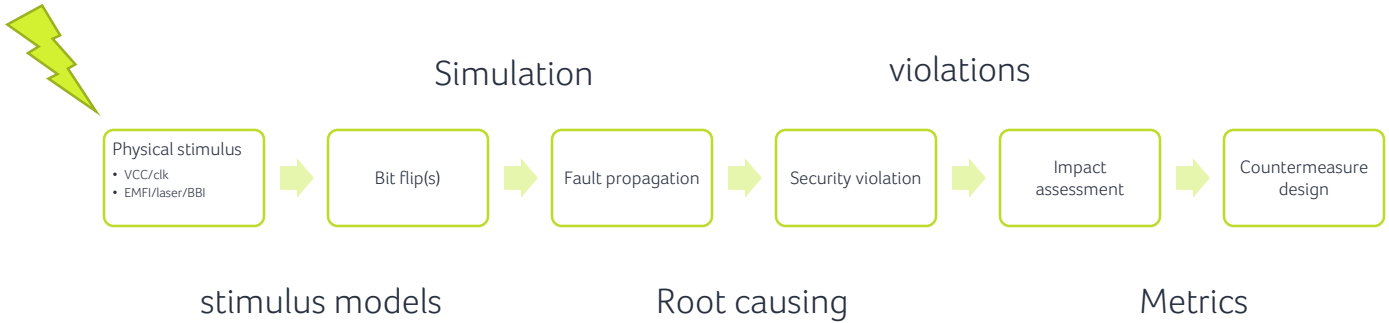
# FAULT SIMULATION ABSTRACTIONS

| Abstraction | Effect |
|---|---|
| Instruction level | Instruction skip, opcode flip (CPU only)<br>https://github.com/Riscure/FiSim |
| ISA level | Instruction opcode change, Memory, register (CPU only)<br>https://hal.archives-ouvertes.fr/hal-03248430/document (CELTIC)<br>https://ieeexplore.ieee.org/document/9565584 (ARCHIE)<br>Riscure FIRM |
| Microarchitecture level | ALU, address generation, decode issues (CPU only) |
| HDL level | Flip a register value<br>Synopsys Zoix, Cadence Xcelium, Mentor Questa, Riscure vectorized FI simulator |
| Cell level | Flip the input/output of a standard cell<br>Riscure parallel FI simulator<br>Same as above |
| Transistor level | Flip the input/output of a transistor |
| Analog / SPICE / FEM | EM field / voltage spike that kicks a transistor over the threshold voltage<br>Setup/hold issues on a FF |

Higher abstraction, faster computation, lower accuracy

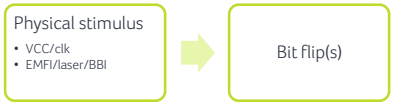
publi

# FI FLOW


Simulation
violations
Physical stimulus
• VCC/clk
• EMFI/laser/BBI
Bit flip(s)
Fault propagation
Security violation
Impact assessment
Countermeasure design
stimulus models
Root causing
Metrics

# RESEARCH: MODEL FAULT STIMULI

- Single bit flips represent “microprobing” attacker, **not** realistic representation of VCC, clock, EM, optical, BBI faults
- Model probability of bit flip:
  - Of a particular **gate output**
  - For a given **circuit** at a particular point in **time** and **space**
  - For a given fault **medium**, probe **size**, **amplitude**, etc
- Create a ‘superset’ model that captures all realistic attacker capabilities
- Note: we have **non-**peer-reviewed data that suggests >90% of multi-bit faults can be explained by single bit flips



# CPU CASE STUDY: PICORV32

**Identify sensitive RTL registers**

Building a scalable tool for hw/sw developers

Can we perform FI in simulation and give actionable results to devs?

- Use picorv32 core
- Brute-force all single flips for all clocks and all registers/wires

https://github.com/YosysHQ/picorv32

```c
void secureboot(void) {
    set_test_status(TEST_START);
    debug("branch test\n");

    set_test_status(TEST_TRIGGER_UP);
    char success = *(volatile unsigned char *)(UART0_BASE_ADDR
    if(success) {
        set_test_status(TEST_TRIGGER_DOWN);
        debug("success\n");
        set_test_status(TEST_FI_SUCCESS);
    } else {
        set_test_status(TEST_TRIGGER_DOWN);
        debug("failure\n");
        set_test_status(TEST_FI_FAIL);
    }
}
```

```
VM000 has 1 glitches at clock 67 for 1 cycles on pins 1531
prerun | machines: 1 start: 0 end: 67
VM000 pc:    00000000
VM000 pc:    00000044
Allocating 0x10000000 bytes of memory for block 0x2. Addr = 0x002ffffc
VM000: trigger for test status to 80 is set
VM000: setting test status to 80
VM000 pc:    00000020
glitch | machines: 1 start: 67, clocks: 800 end_check: -1
VM000: Inserting glitches at cycle 67
VM000 pc:    00000024 (at fault)
VM000 pc:    000000a8
VM000: trigger for test status to 83 is set
VM000: setting test status to 83
VM000 pc:    00000020
VM000 pc:    00000030
VM000 pc:    00000030
VM000 pc:    00000030
VM000 pc:    00000030
VM000 pc:    00000030
VM000 pc:    00000030
VM000 pc:    00000030
VM000 console: success
VM000 pc:    000000c4
VM000: trigger for test status to 40 is set
VM000: setting test status to 40
VM000 Handle tick done, test done from sw; newstatus 40
Test done at cycle 352
```

# RAW RESULTS

- Total faults injected: 1386400
- Exploitable: 408
- First 6:

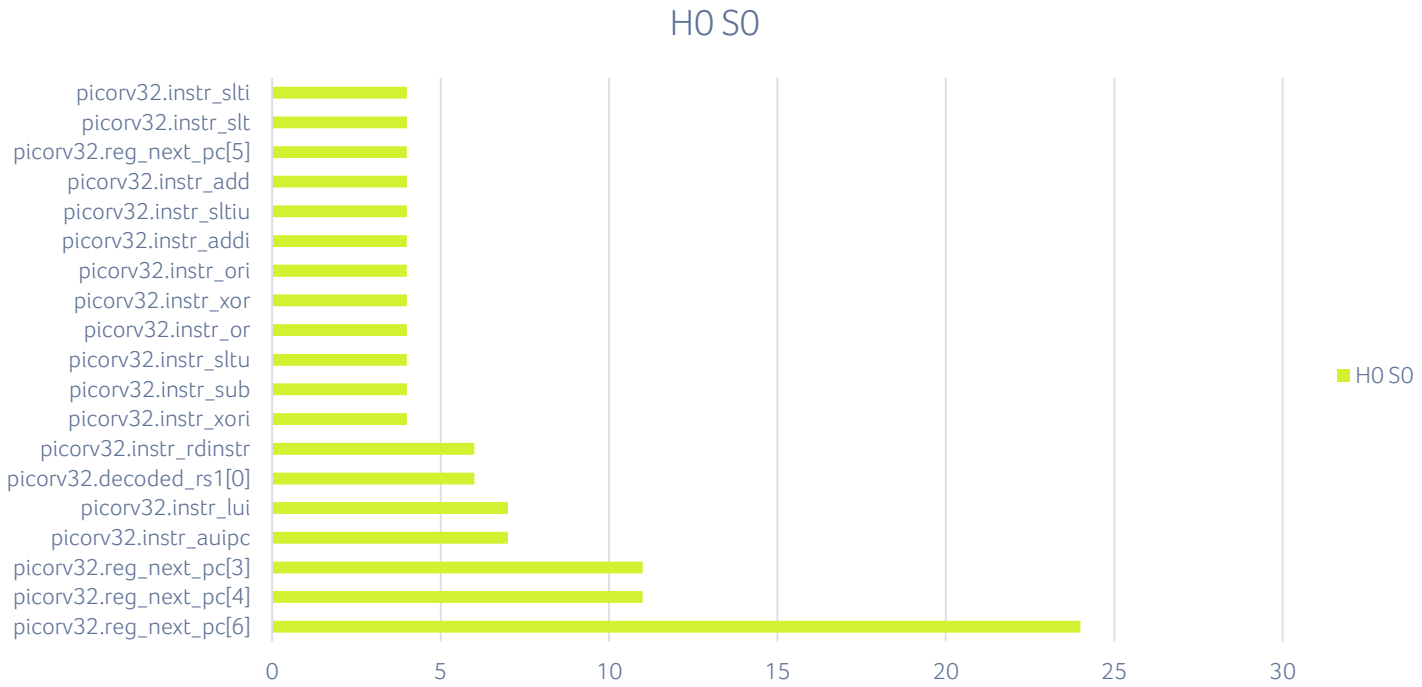
| start | width | clocks | signals | status | pc | signals_name | function | sourcefile |
|---|---|---|---|---|---|---|---|---|
| 0 | 1800 | 60 | TEST_EXPLOITABLE | 00000078 | picorv32.instr_rdcycleh | set_test_status | main.c:38 |
| 0 | 1800 | 61 | TEST_EXPLOITABLE | 00000078 | picorv32.instr_rdinstr | set_test_status | main.c:38 |
| 0 | 1800 | 62 | TEST_EXPLOITABLE | 00000078 | picorv32.instr_rdinstrh | set_test_status | main.c:38 |
| 1 | 1800 | 60 | TEST_EXPLOITABLE | 00000078 | picorv32.instr_rdcycleh | set_test_status | main.c:38 |
| 1 | 1800 | 61 | TEST_EXPLOITABLE | 00000078 | picorv32.instr_rdinstr | set_test_status | main.c:38 |
| 1 | 1800 | 62 | TEST_EXPLOITABLE | 00000078 | picorv32.instr_rdinstrh | set_test_status | main.c:38 |

# RESEARCH: SIMULATION PERFORMANCE

- Single bit flips can be bruteforced in simulation
  - E.g. crypto IP core, “simple” CPU cores
  - Order of days on ‘beefy workstation’
  - ~linear in gates and quadratic in clocks
- We leverage bit-slicing with Verilator to simulate up to 256 netlists in parallel
  - Outperforms non-bitsliced RTL simulation
  - Functional safety simulators do this also, but limit cores by licensing model
- Cloud-scaling with CPU is straightforward
- GPU or FPGA acceleration promises orders of magnitude faster speeds
- How to efficiently leverage FPGA/GPU is open research
- Limit the simulations to the cases interesting for security
  - Formal methods, taint analysis, finding statistical invariants etc., can also help reduce this space.
- Metric: #hours to simulate all relevant faults on a SoC within x h

VERILATOR

# TOP SIGNALS SENSITIVE TO FAULTS




- Reg_next_pc particularly vulnerable
- NB: ‘instruction skip’ or ‘opcode change’ can’t do this

public

# ACTION FOR HW DEVELOPER: SHADOW PC

Duplicate the PC registers, instruct synthesis to keep duplicates

```
-    reg [31:0] reg_pc, reg_next_pc, reg_op1, reg_op2, reg_out;
+     (* keep = 1 *)
+    reg [31:0] reg_pc_d, reg_pc_q, reg_pc2_d, reg_pc2_q, reg_ne
xt_pc_d, reg_next_pc_q, reg_next_pc2_d, reg_next_pc2_q;
```

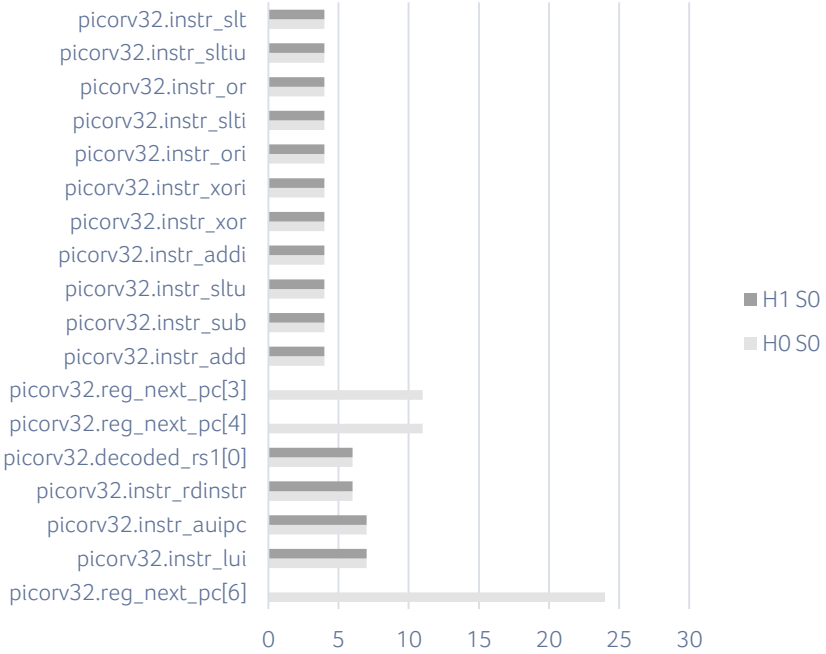
Reset core if the registers differ

```
+        if (!resetn || reg_next_pc_q != reg_next_pc2_q ||
reg_pc_q != reg_pc2_q) begin
+            reg_pc_d = PROGADDR_RESET;
+            reg_pc2_d = PROGADDR_RESET;
```

# CPU CASE STUDY: PICORV32

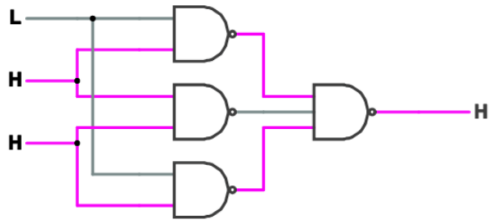
- Identified next_pc as vulnerable, and hardened it
- 4 less exploitable signals leads to 50 less exploitable faults

| | Vanilla | Hardened |
|---|---|---|
| Total | 1386400 | 1540800 |
| Exploitable faults | 408 | 358 |
| Exploitable signals | 268 | 264 |



# RESEARCH: AUTOMATED NETLIST HARDENING

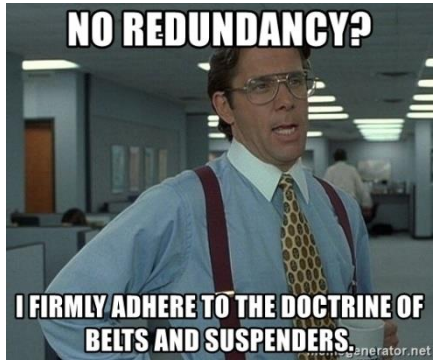
- With a list of vulnerable regs/wires, we can use TMR to mitigate single-bit faults
- Can use ranking of list to trade off security / overhead
- How effective is this (or other local countermeasure) in a circuit?
- Metric: FI resistance before vs FI resistance after



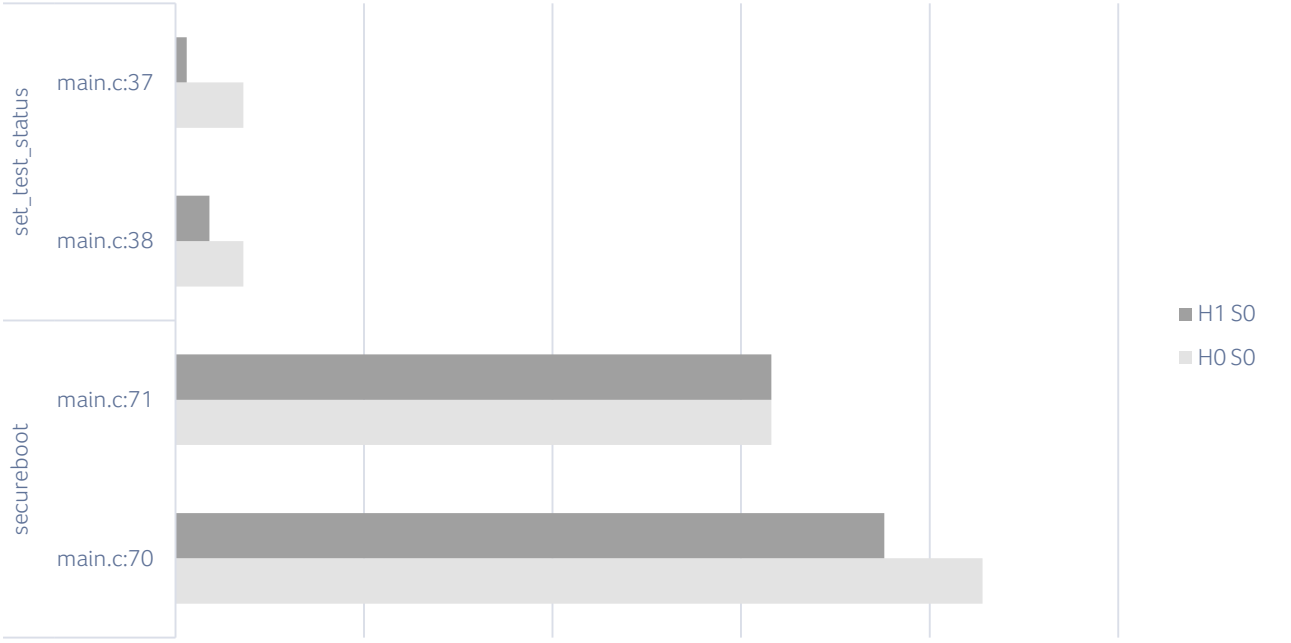

https://en.wikipedia.org/wiki/Triple_modular_redundancy

# ACTION FOR SW DEVELOPER: ADD DOUBLE CHECK

- Hardened code from before
- Run branch test code with double check
- Brute-force all single bit flips



# WHAT LINES OF CODE ARE VULNERABLE?




```
70      char success = *(volatile unsigned char *)(UART0_BASE_ADDRESS + UART0_RX_DATA);
  1     if(success) {
```

```
70      char success = *(volatile unsigned char *)(UART0_BASE_ADDRESS + UART0_RX_DATA);
 1      if(success) {
 2  #if SOURCE_HARDEN
 3          success = *(volatile unsigned char *)(UART0_BASE_ADDRESS + UART0_RX_DATA);
 4          if(success)
 5  #endif
 6          {
 7              set_test_status(TEST_TRIGGER_DOWN);
 8              debug("success\n");
 9              set_test_status(TEST_FI_SUCCESS);
10  #if SOURCE_HARDEN
11          } else {
12              set_test_status(TEST_TRIGGER_DOWN);
13              debug("FI detected\n");
14              set_test_status(TEST_FI_DETECTED);
15  #endif
16          }
17      } else {
18          set_test_status(TEST_TRIGGER_DOWN);
19          debug("failure\n");
20          set_test_status(TEST_FI_FAIL);
21      }
```

Add a double check in software

And an FI detection!

# RESEARCH: AUTOMATIC SOFTWARE HARDENING

- Shown on a **toy example** we can rewrite binaries and inject FI countermeasures **only** in the **vulnerable spots** for **certain opcode faults**
- Open questions:
  - Countermeasure design (next slide)?
- Metric: effectiveness on production code?
- Metric: effectiveness with physical FI?

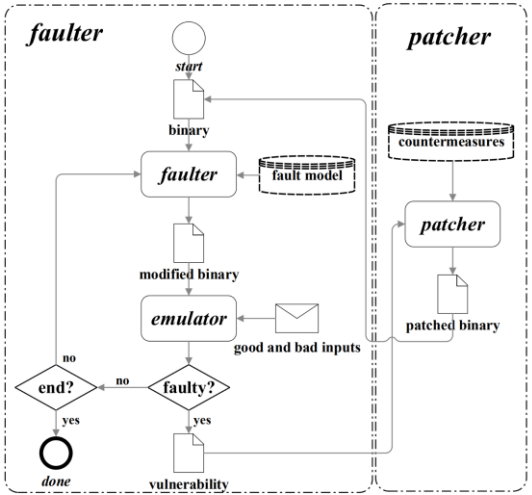



Fig. 2: Flowchart of the *Faulter+Patcher* approach

WPI+Riscure DAC publication: https://ieeexplore.ieee.org/document/9586278
Paper: https://arxiv.org/abs/2011.14067

# RESEARCH: AUTOMATIC SOFTWARE HARDENING

```
cmp rbx, [rcx+4]
fallthrough: ...
```

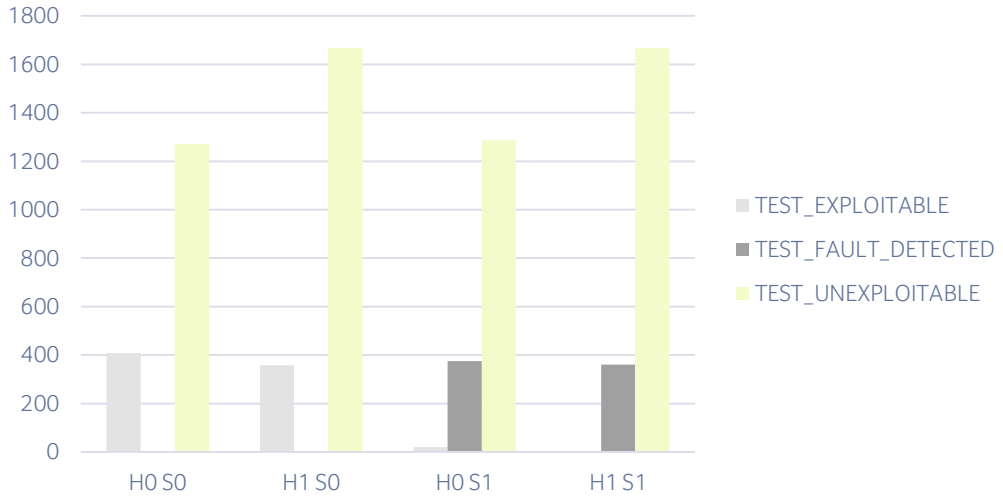


```
lea rsp,[rsp-128]
    cmp rbx, [rcx+4]
    push rbx
    pushfq
    cmp rbx, [rcx+4]
    pushfq
    pop rbx
    cmp rbx, [rsp]
    je restore
    call faulthandler
restore:
    popfq
    pop rbx
    lea rsp,[rsp+128]
fallthrough: ...
```

- Hard problems:
  - Preserve “volatile” semantics
  - Avoid introducing new FI issues
  - SCA?

# CPU CASE STUDY: PICORV32

| | Vanilla | Harden RTL | Harden sw | Harden RTL + sw |
|---|---|---|---|---|
| Total | 1386400 | 1540800 | 1386400 | 1540800 |
| Exploitable faults | 408 | 358 | 20 | 2 |
| Exploitable signals | 268 | 264 | 4 | 2 |
| Detections | 0 | 0 | 376 | 361 |




(this experiment is waaay to limited to conclude anything general about hardware vs software countermeasures)

# RESEARCH: CPU SECURITY VIOLATIONS

- We have a set of CPU test cases covering register, branch, ALU and memory instructions
- “Test a CPU without specific software”
- Open questions:
  - Does this cover a large set of relevant faults for arbitrary software?
  - Can the results of this inform a compiler/binary rewriting tool to harden code?
- Metric: FI resistance metric for CPU
- Riscure publication soon…

```
// test bne
"check_0:\n"
"bne a1, a3, check_1\n"
"j fi_success\n"
// test beq
"check_1:\n"
"beq a2, a3, check_2\n"
"j fi_success\n"
// test blt
"check_2:\n"
"blt a1, a3, check_3\n"
"j fi_success\n"
// test bge
"check_3:\n"
"bge a3, a1, check_4\n"
"j fi_success\n"
// test bltu
"check_4:\n"
"bltu a2, a4, check_5\n"
"j fi_success\n"
// test bgeu
"check_5:\n"
"bgeu a3, a1, check_6\n"
"j fi_success\n"
```

Public

# RESEARCH: SECURITY VIOLATIONS

- Currently designed / typed in by humans
- Riscure has (unpublished) work on automatically validating all known FI issues on AES cores
- CPU cores: FIRM

- “Isadora: Automated Information Flow Property Generation for Hardware Designs” Deutschbein et al

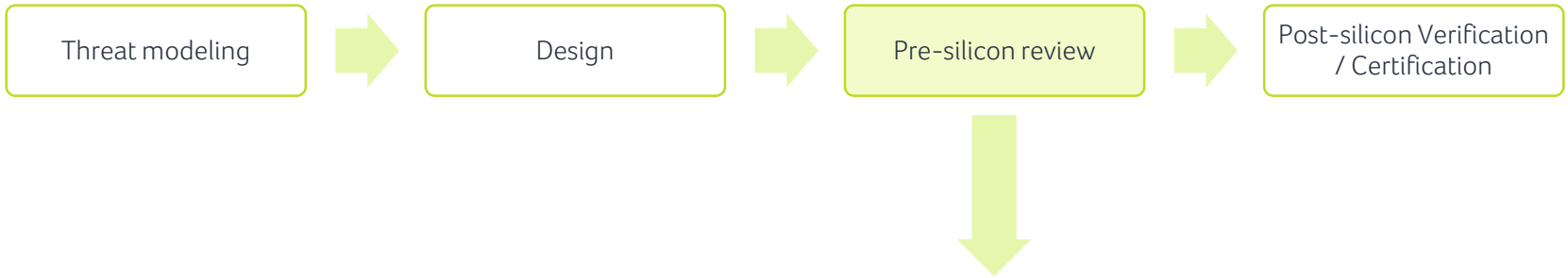
(https://kastner.ucsd.edu/wp-content/uploads/2022/05/admin/ashes21-isadora.pdf )

# LIMITATIONS

- Speed limited by CPU cores / design size
- Simulations are digital only

- FI: focus on single bit flips: good for coverage, though may overestimate attacker capabilities

# CPU CASE STUDY: PICORV32




- Reducing FI was as trivial as adding redundancy
- Re-running tool allows validating various countermeasure options

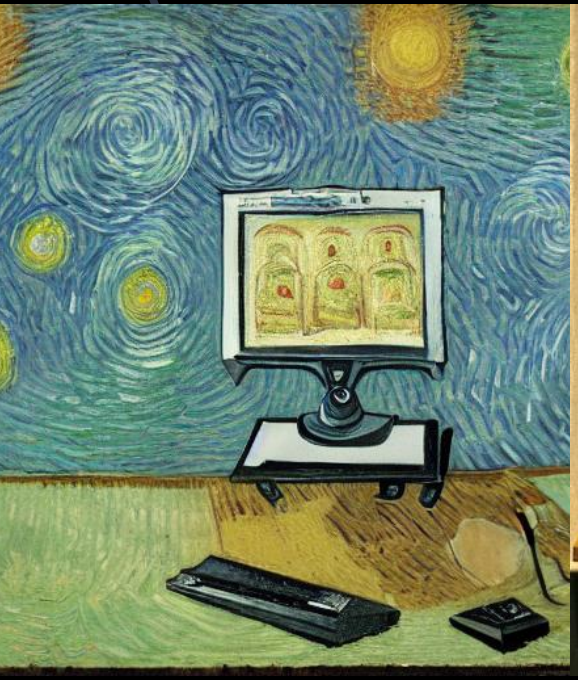
# CONCLUSION

public

Simulation is automated, after initial setup

Simulation can be done during development, by developers

Simulation can show root causes, allowing (automated) fixes



Many open questions!

Email jasper@riscure.com if you want to chat research collaborations / internships, etc

**Riscure B.V.**
Frontier Building, Delftechpark 49
2628 XJ Delft
The Netherlands
Phone: +31 15 251 40 90
www.riscure.com

**Riscure North America**
550 Kearny St., Suite 330
San Francisco, CA 94108 USA
Phone: +1 650 646 99 79
inforequest@riscure.com

**Riscure China**
Room 2030-31, No. 989, Changle Road,
Shanghai 200031
China
Phone: +86 21 5117 5435
inforcn@riscure.com

riscure

driving your security forward