Detecting Hardware Security Vulnerabilities

Jayden Koh, Chen Chen, Jeyavijayan Rajendran
SETH Lab, Texas A&M University

Statement of Problem & Justification

Research Question

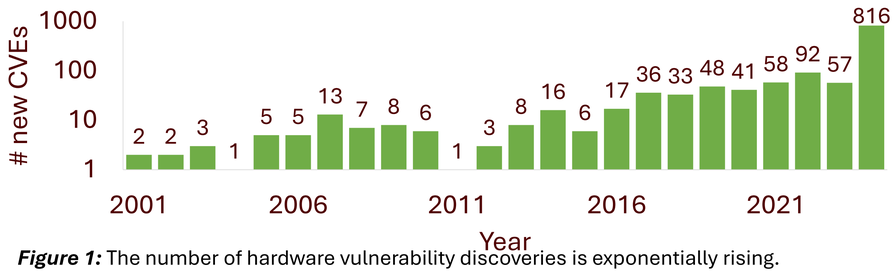


Figure 1: The number of hardware vulnerability discoveries is exponentially rising.

Hardware vulnerabilities are **emerging at an alarming rate**.

Hardware vulnerabilities **are difficult to patch**.

Current hardware security measures are **insufficient** in catching bugs at the pre-silicon stage.

Existing Method	Pros	Cons
Formal Verification	Exhaustively Secure	Not scalable
Security Auditing	Context-Aware	Manually intensive
Directed Tests	Automated	Lack of coverage
Information Flow Tracking	Targeted coverage spaces	False positives

Research Objective

- Analyze the hardware design of several popular cores: **RSD**, **XiangShan**, and **CVA6.**
- Use novel hardware fuzzing techniques to identify functional vulnerabilities.
- Bridge the gap between **industrial verification flow** and academic research.

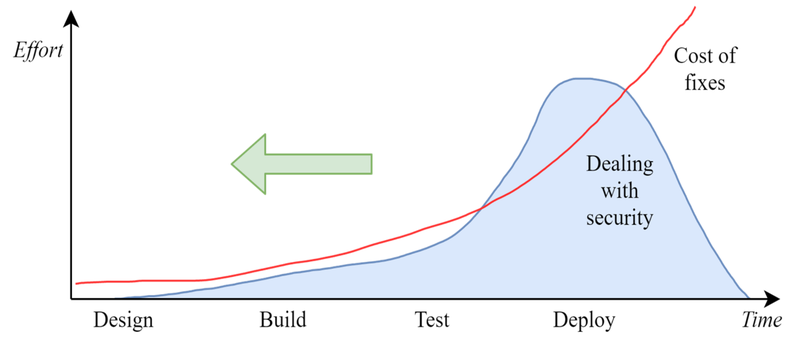


Figure 2: Discovering hardware bugs in the design stage decreases the costs of maintaining and patching processors.

Study Design & Methods

Study Design

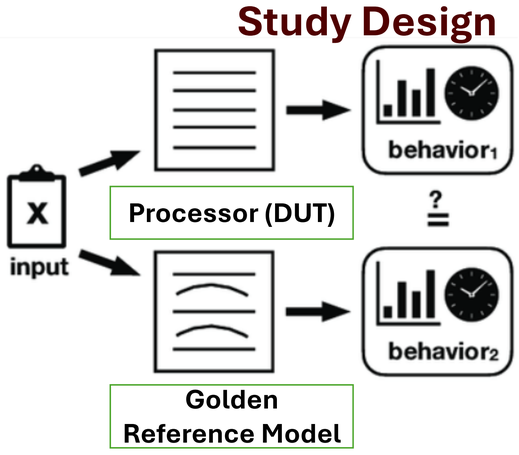


Figure 3: **Differential fuzzing** is a technique for detecting functional bugs in software.

Software differential fuzzing techniques were adapted to compare trace logs from **hardware** simulations. Differences in trace logs revealed **functional bugs** as the processor was not performing to the specification of the official ISA.

Differences from existing hardware verification methods:

Increased **Scalability**	✅
Increased **Automation**	✅
Increased **Coverage**	✅
Decreased **False Positives**	✅

Methodology

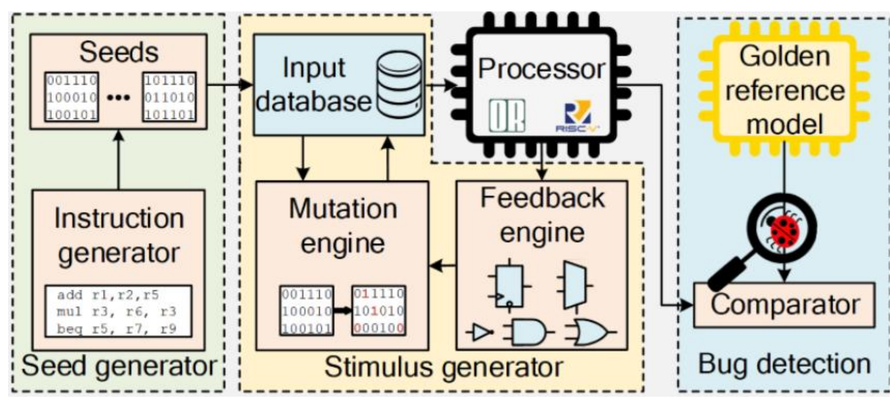


Figure 4: **TheHuzz** framework for hardware fuzzing.

Data Analysis & Interpretation

Data

One of the vulnerabilities found with hardware fuzzing is described below:

```
// GROUP 2.1: set MPP field in mstatus to 01 Supervisor
asm volatile ( "CSRR t0, mstatus" );
asm volatile ( "LI t1, 0xFFFFF7FF" );
asm volatile ( "AND t0, t0, t1" ); // set MPP to 01
asm volatile ( "CSRW mstatus, t0" );

// GROUP 2.2: NOP instructions
asm volatile ( "ADDI x0, x0, 0" );
asm volatile ( "ADDI x0, x0, 0" );
asm volatile ( "ADDI x0, x0, 0" );

// GROUP 2.3: switch to supervisor mode
asm volatile ( "LA t0, supervisor_mode_code" );
asm volatile ( "CSRW mepc, t0" ); // set mepc to supervisor_mode_code
asm volatile ( "MRET" );

// GROUP 2.4: NOP instructions
asm volatile ( "ADDI x0, x0, 0" );
asm volatile ( "ADDI x0, x0, 0" );
asm volatile ( "ADDI x0, x0, 0" );

// GROUP 2.5: supervisor mode code with writes to SRO CSRs
asm volatile ( "supervisor_mode_code:" );
asm volatile ( "LI t0, 0x12345678" );
asm volatile ( "CSRW 0xDA0, t0" );  // scountovf
```

Figure 5: Contents of an assembly exploit that show the unexpected behavior with supervisor privilege mode.

Figure 6: **Simulation waveform** showing the processor switching privilege contexts from Machine to Supervisor.

```
00001bb0 <supervisor_mode_code>:
    1bb0:       123452b7                lui     t0,0x12345
    1bb4:       67828293                addi    t0,t0,1656 #
    1bb8:       da029073                csrw    scountovf,t0
```

Figure 7: Executable binary showing an illegal instruction in supervisor-mode.

```
      5261,0x1bb0,0x00000000,0x00001010,
0,0x00000000,0x00000000,0x00000000,0x0000
000000,0x00000000,0x00000000,0x00000000,0
,0x00000000,0x00000000,0x00000000,0x0000
      5262,0x1bb4,0x00000000,0x00001010,
0,0x00000000,0x00000000,0x00000000,0x0000
000000,0x00000000,0x00000000,0x00000000,0
,0x00000000,0x00000000,0x00000000,0x0000
      5278,0x1bb8,0x00000000,0x00001010,
0,0x00000000,0x00000000,0x00000000,0x0000
000000,0x00000000,0x00000000,0x00000000,0
,0x00000000,0x00000000,0x00000000,0x0000
```

Figure 8: Program does not crash on illegal instruction exception.

Results

The vulnerability regarding **failure to throw illegal instruction exception** is described with a simulation waveform for visual verification, trace logs for automated verification, and an assembly exploit for replicating the vulnerability.

A total of **6 zero-day** bugs were discovered in the RSD processor including:

- Unvalidated **FS** flag
- Unhandled **FS** flag
- Unhandled **SD** flag
- Machine writable **SD** flag
- User readable **MPP** flag
- Unhandled writable **MPP** flag

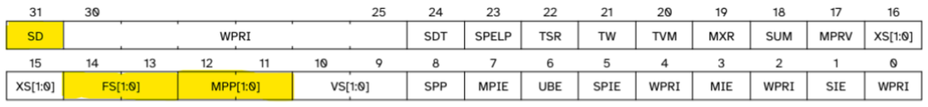


Figure 9: *Affected fields in the mstatus CSR register on the RSD processor.*

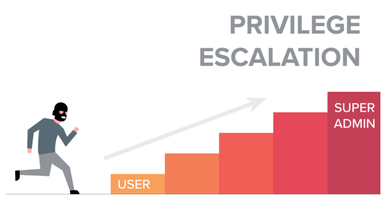
All bugs were responsibly and **ethically disclosed** by providing a **technical writeup** to the product's authors and a **high-level overview** to their respective security authorities for official acknowledgement.

Discussion & Conclusion

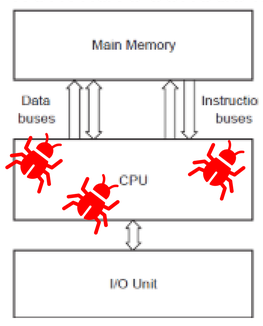
Conclusion

Each individual bug discovered from this work could lead to:

- **Privilege escalation**
- **Isolation bypassing**
- **Silent failures**



With an average recommended CVSS v4.0 score of 9.3/10 "**Critical**", the bugs found in this work have serious implications on the entire execution process of the RSD processor.



These results show that hardware fuzzing is an effective method of verifying the architectural security of open-source processors.

Additionally, the discovery of such **severe implementation flaws** of the RISC-V ISA demonstrates the insufficiency of current hardware verification practices. These findings highlight the urgent need for the integration of a complete hardware fuzzing validation suite to mitigate the potential for **critical architectural exploits** that other methods might miss.

Further Research

Further research will be conducted to determine:

- Efficacy of hardware differential fuzzing across **other processors** (XiangShan, CVA6S+)
- **Root cause** of the 6 RSD vulnerabilities within the source code
- Optimizing input generation and mutations to **increase search space** and coverage
- Integration with existing hardware verification methods as a **hybrid solution**
- Reducing **false positives** in the bug detection process