Microscope: Causality Inference Crossing the Hardware and Software Boundary from Hardware Perspective

Zhaoxiang Liu
Kansas State University
Manhattan, KS, USA
zxliu@ksu.edu

Kejun Chen
Kansas State University
Manhattan, KS, USA
kejun@ksu.edu

Dean Sullivan
University of New Hampshire
Durham, NH, USA
dean.sullivan@unh.edu

Orlando Arias
University of Massachusetts Lowel
Lowell, MA, USA
orlando_arias@uml.edu

Raj Dutta
Silicon Assurance
Gainesville, FL, USA
rajgautamdutta@siliconassurance.com

Yier Jin
University of Science and Technology of China
Hefei, Anhui, China
jinyier@ustc.edu.cn

Xiaolong Guo
Kansas State University
Manhattan, KS, USA
guoxiaolong@ksu.edu

Abstract

The increasing complexity of System-on-Chip (SoC) designs and the rise of third-party vendors in the semiconductor industry have led to unprecedented security concerns. Traditional formal methods struggle to address software-exploited hardware bugs, and existing solutions for hardware-software co-verification often fall short. This paper presents Microscope, a novel framework for inferring software instruction patterns that can trigger hardware vulnerabilities in SoC designs. Microscope enhances the Structural Causal Model (SCM) with hardware features, creating a scalable Hardware Structural Causal Model (HW-SCM). A domain-specific language (DSL) in SMT-LIB represents the HW-SCM and predefined security properties, with incremental SMT solving deducing possible instructions. Microscope identifies causality to determine whether a hardware threat could result from any software events, providing a valuable resource for patching hardware bugs and generating test input. Extensive experimentation demonstrates Microscope's capability to infer the causality of a wide range of vulnerabilities and bugs located in SoC-level benchmarks.

Keywords

Causality Inference, Hardware Security, Hardware and Software Co-Verification

ACM Reference Format:
Zhaoxiang Liu, Kejun Chen, Dean Sullivan, Orlando Arias, Raj Dutta, Yier Jin, and Xiaolong Guo. 2023. Microscope: Causality Inference Crossing the Hardware and Software Boundary from Hardware Perspective. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

System-on-chip (SoC) designers face unprecedented security concerns due to the rise of third-party vendors in the semiconductor industry [16]. As the complexity of SoC designs rises, securing a computer system requires a thorough understanding of the full software stack and the hardware architecture. SoC designers have been overwhelmed with the workload of manually diagnosing security vulnerabilities. Moreover, software-exploited hardware bugs present rigorous challenges to traditional formal methods. The emergence of transient execution attacks [5] has propelled the topic of software-hardware co-verification into the limelight. However, existing work falls short when it comes to bridging the gap between hardware and software.

For example, Coppelia [21] generates software exploit for hardware bug to help engineer contextualize the threat. This work migrates hardware language to the software platform to address the co-verification issue. Specifically, the hardware Register Transfer Level (RT-level) design is firstly translated to C++ and then takes advantage of KLEE [3] for symbolic execution. Based on the security-critical assertion. The violation input sequence can be found for exploit generation. However, the software-level descriptions of hardware cannot precisely represent the behavior of the original RT-level code. This abstraction can pose a challenge to accurately analyze the structures within the produced C++ code. Moreover, micro-architectural design can be verified as an RT-level design using open-source EDA tools such as Yosys [20], commercial EDA tools like Cadence JasperGold [4], and Synopsys VC Formal [17]. However, these tools are not originally designed for hardware-software co-verfication. While engineers can still use assertion-based verification techniques to validate potential threats, doubts remain about uncovering all interaction traces between software and hardware.

In response to these challenges, this paper presents Microscope, an innovative framework designed to infer potential software instruction patterns that expose hardware vulnerabilities. Specifically, we enhance the Structural Causal Model (SCM) [9] with hardware features such as timing stamps, resulting in a scalable Hardware Structural Causal Model (HW-SCM). A domain-specific language (DSL) in SMT-LIB is developed to represent this HW-SCM along with predefined security properties. Subsequently, incremental SMT solving is applied to deduce all possible instructions that satisfy these properties. The effectiveness of Microscope is validated in several RISC-V SoC benchmarks. The primary contributions of this paper are as follows.

- The proposed Microscope presents HW-SCM in a self-developed DSL, which models a hardware-software system from the hardware side. It effectively addresses scalability and efficiency issues by eliminating the need to convert hardware to software or low-level hardware description.
- Our approach identifies causality to determine whether a hardware threat could result from any software events. If such a relationship exists, the inferred instruction patterns can serve as a valuable resource for both patching hardware bugs and generating test input. Conversely, if no such relationship is found, the hardware vulnerability can be deemed a non-critical security threat, allowing for the conservation of resources.
- Through extensive experimentation, we demonstrate that Microscope is capable of inferring the causality of a wide range of vulnerabilities and bugs found in SoC-level benchmarks. In line with this, we have developed a corresponding EDA tool that will be made publicly available upon the paper's acceptance.

2 Background

2.1 Structural Causal Model

A Structural Causal Model (SCM) is a mathematical framework used to describe the causal relationships between variables in a system [9]. Two set, U and V, alongside a function set defined as

$$f = \{f_x : W_x \rightarrow x | x \in V\}, \tag{1}$$

where $W_x = U \cup V \setminus \{x\}$. V is referred to as a set of *endogenous variables* or variables within the SCM, and U is a set of *exogenous variables* or variables external to the SCM. As such, W_x is the union of all variables present in the design with the exception of x. Every endogenous variable is a descendant of a subset of exogenous variables as defined by f_x, and $\forall y \in W_x$, y is a cause of x.

2.2 SMT Solving.

Satisfiability Modulo Theories (SMT) solving is a powerful approach for automated verification and constraint satisfaction in software and hardware design, which supports a wide range of theories, including quantifier-free bit-vector (QF-BV), linear integer arithmetics (LIA), etc. Quantifier-Free Bit-Vector Logic (QF-BV) supports reasoning about fixed-size bit-vectors and their operations without the use of quantifiers such as Bit-wise Logical Operations, Arithmetic Operations, etc. In this paper we use a subset of QF-BV Operations to encode the verilog into HW-SCM format.

3 Hardware Structural Causal Model

This section delineates the HW-SCM as a multi-layer graph model that facilitates causality inference at the hardware-software boundary. We initiate this discourse by elucidating the concept of the HW-SCM and its correlation with the traditional SCM and hardware design.

3.1 HW-SCM definition

HW-SCM extends the foundational concept of SCMs [9], applying it to model software as a sequence of signals within the hardware schematic. In this context, we assume that SW_i represents the set of instructions or input signals in Clock i, while HW_i represents the set of hardware signals (excluding the inputs) in Clock i ($i \in \mathbb{N}$). To characterize the HW-SCM, we define two sets of functions:

$$f_{comb} = \{f_i : X_i \rightarrow y_i \mid y_i \in HW_i\}, \tag{2}$$

$$f_{seq} = \{f_i : X_{i-1} \rightarrow y_i \mid y_i \in HW_i\}, \tag{3}$$

Here, $X_i \subseteq (SW_i \cup HW_i) \setminus \{y_i\}$ represents a subset of signals, excluding signal y_i, from the combined set of software and hardware signals. The set f_{comb} encompasses all combinational connections within the design, while f_{seq} encompasses all sequential logic such that every cause of signal y_i, denoted as x_{i-1}, belongs to the clock cycle $i-1$.

Following the traditional SCM model, we consider instructions and inputs to be exogenous variables as these are external stimulus to the hardware, and signals internal to the hardware to be endogenous variables since their state is a direct response to external stimulus.

3.2 HW-SCM Graph Example

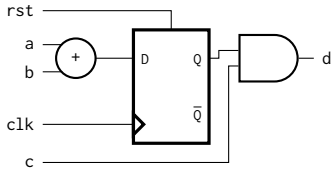
To illustrate the HW-SCM multi-layer graph model, we present an example as shown in Figure 1. The Verilog code is provided in Figure 1a, with its netlist representation given in Figure 1b. In this example, the output signal `d` depends on both signals `e` and `c`. The signal `c` is an I/O port, while signal `e` is updated by inputs `a` and `b` at the positive edge of the clock signal (`clk`). The graph representation of SCM in Figure 1c illustrates the signal dependencies within the design. However, since SCM is not specifically designed for hardware, it may fall short in accurately modeling certain hardware behaviors, such as timing behavior. For instance, SCM is not capable of adequately addressing delay propagation in sequential circuits.

Therefore, we develop the HW-SCM by introducing two function sets: f_{comb} and f_{seq} to fit the hardware domain-specific modeling. These function sets define different types of edges in the graph model, mapping hardware signal connections to the multi-layer structure depicted in Figure 1d. HW-SCM is represented as a multi-layer graph, where f_{comb} represents the connections within a layer, capturing the combinational dependencies, and f_{seq} represents the connections across layers, capturing the sequential dependencies. Hence, the hardware timing behavior can be identified through this multi-layer graph representation. Each layer represents a hardware state space within one clock cycle.

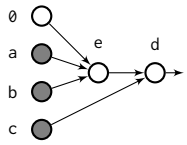
In Figure 1d, we present two layers: *Clock i* and *Clock i-1*. In the *Clock i* layer, the output signal d_i depends on the value of signals c_i and e_i at the same clock cycle, *Clock i*. The signal e_i in the *Clock i* layer is updated on the positive edge of the clock signal (`clk`) based on the inputs a_{i-1} and b_{i-1} at the previous clock cycle, *Clock i-1*. This means that the value of d_i at *Clock i*

```
module top (
        input a, b, c,
        input clk, rst,
        output d);
    reg e;
    assign d = e & c;
    always @(posedge clk
            or posedge rst)
        if(rst) e <= 1'b0;
        else e <= a + b;
endmodule
```

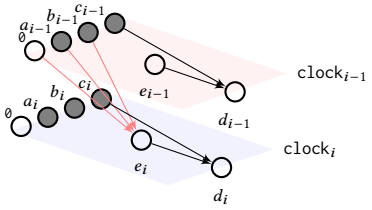
(a) Verilog example.



(b) Circuit graph example.



(c) Graph representation of SCM.



(d) Graph of HW-SCM.

Figure 1: Graph based HW-SCM

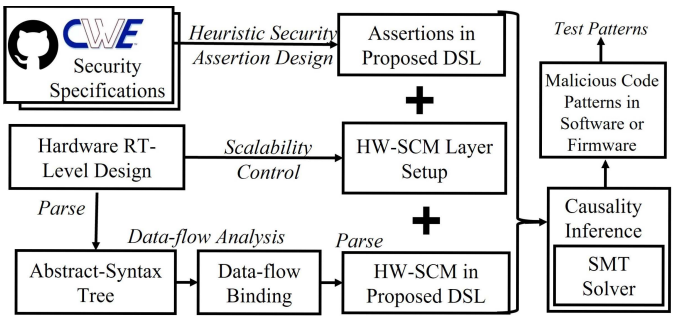
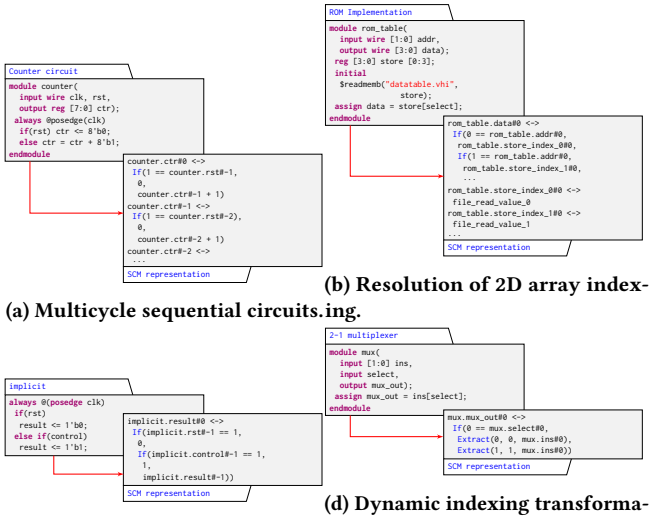


Figure 2: Working procedure of the proposed Microscope framework



(a) Multicycle sequential circuits.

(b) Resolution of 2D array indexing.

(c) Implicit `else` transformation.

(d) Dynamic indexing transformation.

Figure 3: Domain Specific Language

is determined by the inputs a_{i-1} and b_{i-1} at the previous clock cycle, *Clock i-1*, as well as the input c_i at the current clock cycle, *Clock i*. The hardware system within two consecutive time slots can be exemplified using this two-layer HW-SCM. By extending the HW-SCM into an N-layer model, we can capture the multi-clock cycle behavior of the hardware system within consecutive time slots. Moreover, the sequences of instructions from the software will be modeled as input signals with consecutive timestamps in HW-SCM. This allows for a more comprehensive representation and analysis of the hardware-software system's functionality and temporal dependencies.

4 Microscope

Microscope uses the HW-SCM as its infrastructure to enable automated hardware-software co-analysis. This section introduces Microscope and elaborates on the details of its primary steps.

4.1 Microscope Overview

Figure 2 provides a diagram of the general operational procedure of Microscope. The hardware RT-level designs are first translated into HW-SCM. In this process, information-flow tracking (IFT) is employed to traverse the abstract syntax tree (AST), which forms the basis for generating a data-flow graph. HW-SCM is then constructed based on this data-flow graph. We have designed a DSL specifically intended for representing the HW-SCM in SMT-LIB. A heuristic approach is adopted to design/obtain assertions using extant hardware databases like CWE [1] and Bugzilla [21] as references. Simultaneously, it is necessary to determine the number of layers to restrict the scale of the HW-SCM. This figure hinges on the number of clock cycles the user intends to consider for the security assessment. The HW-SCM model, assertions and the number of model layers are represented utilizing the proposed DSL.

Microscope then performs causality inference using an SMT Solver. This inference involves deriving solutions that begin with the assertions at the bottom layer of the HW-SCM model. Solutions consist of same-layer inputs as defined in Equation (2) and higher-layer sequential inputs as defined in Equation (3). Consequently, the inputs from each layer are accumulated and interpreted as instructions patterns that can satisfy the assertion. This code pattern is then documented and blocked in the SMT Solver. Microscope runs this incremental solving process until all code patterns have been inferred. Any returned code patterns can be utilized to generate large-scale test patterns for the design to further explore potential vulnerabilities, as well as to fix hardware bugs.

4.2 Domain-Specific Language for HW-SCM

In the process of converting Verilog source code into HW-SCM SMT representation, several steps need to be followed. Firstly, all variables and parameters present in the source code are identified and treated as bit-vectors. Next, Verilog operators and bit-vector manipulations are mapped to their corresponding counterparts in QF-BV, ensuring the proper translation of operations. Lastly,

control structures such as if-else statements and case statements are transformed into SMT conditional expressions, allowing for logical reasoning and analysis within the Microscope framework.

4.2.1 Modeling Sequential Logic Each signal is assigned a timestamp indicating the clock cycle during which its operation occurs. In HW-SCM, the same hardware signal, when marked with different timestamp values, is treated as distinct symbols. This is demonstrated with timestamps in Figure 3a. `ctr` register is updated every clock cycle, depending on the state of the `rst`. The proposed DSL applies timestamps by adding the `#-n` suffix to signals, where `n` corresponds to a specific clock cycle.

4.2.2 Handling Dynamic Indexing Operations Hardware description languages generally support *bit-selects* and *part-selects*. A bit-select operation occurs when an index from an array of `wire` or `reg` is selected, while a part-select operation happens when multiple consecutive indices are chosen. The proposed DSL introduces an equivalent control structure wherein an `If` condition coupled with statically indexed `Extract` constructs are used to obtain the necessary indices from the array. This construct is extended for all possible indexing values. An example is demonstrated through a simple 2-1 line multiplexer that uses a bit-select operation to set the output, is shown in Figure 3d.

4.2.3 Handling Implicit Control Structures When encountering a `case` statement or an `if-else` statement that does not specify default behavior in a sequential block, the DSL assumes it to retain the value from the previous clock cycle. As depicted in Figure 3c, an implicit `else` statement is added to uphold consistency in modeling the behavior of the circuit within its HW-SCM.

4.2.4 Handling 2D Array Selection and Initialization Constructs The proposed DSL flattened the array into individual elements and used a series of nested `If` statements to represent the indexing operation. Initialization constructs, such as `initial` procedural blocks and externally imported data, are parsed to basic assignment operations in the proposed DSL. The example is showcased in Figure 3b, where a ROM table is inferred and initialized with data from an external file. The resulting conversion breaks down the `store` array into individual elements and assigned to the output based on the value of `addr`. Moreover, individual entries are initialized by reading and parsing the contents of the file specified by the `$readmemb()` function.

4.3 Assertion Development

Microscope utilizes both the HW-SCM (in DSL) and user-specified security assertions (expressed as DSL assertions) to perform causality inference. Microscope users construct assertions using the CWE database for guidance on classes of vulnerabilities that may apply to their design. This is standard practice in design verification as illustrated in [4, 17, 18, 21]. Before constructing the HW-SCM, users are required to specify the time window size, denoted as n, which also reduces scalability issues. Based on the specification, the Microscope will generate an HW-SCM model consisting of $n+1$ layers for verification purposes. This implies that the given assertion will be evaluated within a time span of $n+1$ consecutive clock cycles. Each signal is replicated $n+1$ times with a extended numerical index. For instance, if the original signal is named `signal` the HW-SCM signals will be named `signal#0` (current clock cycle), `signal#-1` (previous clock cycle) up to `signal#-n` (n clock cycles ago) representing the signal value from historical state. Leveraging these annotations, we are able to convert SystemVerilog Assertions (SVA) or Property Specification Language (PSL) into our Domain-Specific Language (DSL) assertions, thereby effectively covering both combinational and sequential assertions.

The SMT solver returns either a *SAT* or *UNSAT* result based on the given assertions and HW-SCM. **Note**: the instruction input signal is also replicated $n+1$ times for recording historical inputs, which are *endogenous variables* in the HW-SCM. Therefore, if the solver returns *SAT*, it means that the given DSL assertion and HW-SCM model allow for a valid stimulus to the system. In other words, there exists a value assignment to the software instructions (causality reason) that can trigger the root of hardware security threats (causality result). Following this, Microscope will employ incremental solving to identify all malicious instruction sequences that could activate the hardware vulnerability.

5 Experimental Results

5.1 Experiment Settings

A series of System-on-Chip (SoC) level evaluations with two distinct Instruction Set Architecture (ISA) SoCs - RISC-V and OpenRISC are performed in this work. Specifically, we selected the DarkRISCV[6], RISC-V mini[19], and OpenRISC 1200[15] processors as the test bench. The vulnerabilities we used for testing are derived from the work[2] and the commit history from OR1200. The results of these experiments are summarized in Table 2. Our testing environment consisted of a machine running Ubuntu 20.04, equipped with an i9-12900K processor and 32GB of memory. Z3 [7] is applied as the SMT solver in this experiment.

5.2 Threat Model and Heuristic Assertions Development

Our framework serves as a valuable tool for verification engineers in finding causality between RT-level vulnerabilities within RT-level designs and software-level instructions. It offers a static method that validates the presence of these vulnerabilities by inferring their input patterns. These input patterns can consist of either compiled or assembled instructions for a processor in the SoC. The experiment demonstrates the Microscope by encompassing three CWE vulnerabilities and five types of design flaws extracted from the OR1200 commit history. In Table 1, we provide a comprehensive overview of these vulnerabilities, including the associated suspicious code patterns within the design that expose them, accompanied by concise descriptions for clarity.

Specifically, CWE-1262 represents an improper access control vulnerability related to a register interface, CWE-1234 relates to hardware internal or debug modes that allow for overriding locks, CWE-1245 pertains to the presence of improper finite state machines in hardware logic. Additionally, Bugzilla #51 and Bugzilla #76 highlight flaws in the ALU design, Bugzilla #90 demonstrates incorrect exception handling, and Bugzilla #88 and Bugzilla #97 exemplify incorrect implementations of instructions. The developed vulnerability assertions used in Microscope are listed in the last column of Table 2 and explained in the following paragraphs.

5.2.1 Bugzilla #51, #76 Two design flaws were identified in the ALU module of OR1200 when performing unsigned comparisons. The problem originates from the incorrect configuration of the `a_lt_b` flag, leading to erroneous computation outcomes. To identify the

Threat	Vulnerability Description	Verilog code Example
CWE-1262 [8], Mail#00007(OR1200) [21]	**Improper Access Control for Register Interface.** The value of the register is not protected, and security-related hardware data could be tampered with through the register interface. Specifically, the write operation targeting the register interface is not adequately validated.	`if (write_en)` `  regfiles[sensitive_index] <= write_data;`

CWE-1234[11]	**Hardware Internal or Debug Modes Allow Override of Locks.** A trusted lock bit, when enabled, can block the write operation to a set of registers or address regions. However, debug mode can bypass the lock mode and modify the locked device configuration, i.e., an external debug signal can override the original lock signal.	`input debug;`
		`/* ... */`
		`if (en || debug)`
		`  peripheral_register <= data;`

CWE-1245[12]	**Improper Finite State Machines (FSMs) in Hardware Logic.** A specific input signal or signal sequence may cause the finite state machine (FSM) in the hardware logic to enter an undefined state, potentially resulting in a denial of service attack or privilege escalation.	`case (opcode)`
		`  7'b0000011: /* incomplete load */`
		`  7'b1101111: /* ... */`
		`  7'b0010111: /* ... */`
		`  default:  /* ... */`
		`endcase`

Bugzilla #51, Bugzilla #76 [21]	**Comparison wrong for unsigned inequality with different MSB.** ALU module yeild incorrect result for unsigned comparation .	`assign a_lt_b = comp_op[3] ?`
		`((a[width-1] & !b[width-1]) | (!a[width-1]`
		`& !b[width-1] & result_sum[width-1])`
		`|(a[width-1] & b[width-1]`
		`& result_sum[width-1])):result_sum[width-1];`

Bugzilla #90[21]	**EPCR on range exception is incorrect.** Exception program counter register doesn't reset to the address of jump instruction before the instruction that caused exception.	`else if (except_trig[13:3] == 11'b1 ) begin` `  except_type <= `OR1200_EXCEPT_RANGE;` ` epcr <= ex_dslot ?` ` wb_pc : delayed1_ex_dslot ?` ` id_pc : delayed2_ex_dslot ?` ` id_pc : id_pc;` ` dsx <= ex_dslot; end`

Bugzilla #88[21]	**l.extw instructions behave incorrectly** No need to explicitly apply an extend operation when using the l.extw instruction.	`case (alu_op)`
		`5'b0_1101:`
		`  result = extended;`

Bugzilla #97[21]	**Ignore an exception that it should handle.** When encountering an unsupported instruction, the control unit should recognize this condition and handle it appropriately, typically by generating an exception or interrupt.	`case (id_insn[31:26])`
		`6'b101110:`
		`  except_illegal <=  1'b0;`

Table 1: Vulnerabilities description and code example

trigger pattern, the assertion specifies the erroneous behavior where the operand `a` is greater than `b` while the `a_lt_b` flag is still set. Microscope traces back the input signal `icpu_dat_i#i`(32 bit instruction) to identify the root cause.

*5.2.2 Bugzilla #90*In the OR1200 processor, when handling a range exception, the exception program counter register (`epcr`) is reset to the jump instruction that was executed prior to the exception. The specific program counter value to be used for the reset is stored in either `dl_pc`, `id_pc`, or `ex_pc`, depending on the delay slot where the exception-causing instruction is located. We track the trigger pattern where the `epcr` is incorrectly set during an exception occurring in the second delay slot,

*5.2.3 Bugzilla #88*When the value of `alu_op` is set to 13 (EXTW), the ALU output is incorrectly updated due to the assignment of the wrong operand. We track the trigger pattern that leads to these incorrect updates of the ALU output.

*5.2.4 Bugzilla #97*OR1200 will not throw an exception when `l.ror` is not implemented. For specified ISA, we track whether one missing instruction can raise the exception.

Benchmark	Vulnerability	Language	Cell Number	Layers	Time	Inference	DSL Property
DarkRISC-V	CWE-1262	Verilog	8,890	3	1.73 s	✓	`reg_wb_addr#0 == 0`
RISCV-mini	CWE-1262	CHISEL	17,108	3	8.16 s	✓	`reg_rf_wen#0 == 1`
DarkRISC-V	CWE-1234	Verilog	8,828	3	2.94 s	✓	`(io_dmem_wen#0 || debug#0) ==1`
RISCV-mini	CWE-1234	CHISEL	17,000	3	24.38 s	✓	
DarkRISC-V	CWE-1245	Verilog	8,971	3	1.45 s	✓	`req_valid#0== 1`
RISCV-mini	CWE-1245	CHISEL	16,927	3	11.36 s	✓	
OR1200	Bugzilla #51	Verilog	20,668	6	23.91 s	✓	`a_lt_b#0==1,comp_op#0[3]==0, a#0>b#0`
OR1200	Bugzilla #76	Verilog	20,714	6	24.03 s	✓	
OR1200	Bugzilla #88	Verilog	20,901	6	23.80 s	✓	`result#0==extended#0, alu_op#0==13`
OR1200	Bugzilla #90	Verilog	20,743	6	18.44 s	✓	`except_trig#-1[13:3] == 1, ex_dslot#-1 == 0, delayed1_ex_dslot#-1 == 1, epcr#0 == id_pc#0`
OR1200	Bugzilla #97	Verilog	20,945	6	18.42 s	✓	`id_insn[31:26]#-1== 46, ex_freeze#-1==0, id_freeze#-1==0, ex_flushpipe#-1==0, except_illegal#0==0`

Table 2: Experiment Result

*5.2.5 CWE-1234*We implement CWE-1234 by altering the control signal of the AES256 peripheral register. The malicious code snippet can be found in the last column of Table 1. The peripheral register value can be updated when the signal `io_dmem_wen` is set or debug mode is turned on. The trigger condition is either `io_dmem_wen == 1` or `debug == 1`. We build the HW-SCM based on the constraint `(io_dmem_wen#0 || debug#0) ==1`. and restrict the time window to 3 in order to backtrack the input pattern that can trigger this assertion.

*5.2.6 CWE-1262*Zero Register is considered a read-only register and cannot be written to in a processor. However, the vulnerability CWE-1262(Mail#00007(OR1200)) applies write permission to the Zero Register, altering its intended behavior. Referring to the Table 1, we build the constraint `reg_wb_addr#0 == 0 && reg_rf_wen#0 == 1` to backtrack the input pattern

*5.2.7 CWE-1245*An undefined state `opcode=7'0000011` is intentionally inserted in the processor decoder as shown in Table 1. Microscope utilizes the constraint `req_valid#0 == 1'b1` to track the opcode pattern that enable the signal.

5.3 Results and Analysis

CWE and Bugzilla vulnerabilities described in Table 1 are utilized to design security assertions in SoC-level benchmarks. We use Microscope to infer instruction patterns in three SoC platforms – RISCV-mini, DarkRISC-V, and OR1200. Microscope support various hardware languages, including Verilog and Chisel. The scalability of the benchmarks is evaluated based on the number of CMOS gates (NAND/NOR) in the column of **Cell Number**. **Layers** represents the number of clock cycles considered in the verification to resolve the assertion. **Time** of Microscope measures the total running time, including both HW-SCM building and Z3 solving.

*5.3.1 Causality Inference in OR1200*Taking Bugzilla #51 and Bugzilla #76 as examples, one of the inference results are shown in Listing 1. It reveals four instructions from SoC instruction memory input `icpu_dat_i` that can trigger the vulnerability assertion. These instructions can incorrectly set the `a_lt_b` flag.

```
icpu_dat_i:
 l.sfgtu  rA,rB   //11100100010...
 //or
 l.sfgeu  rA,rB   //11100100011...
 //or
 l.sfltu  rA,rB   //11100100100...
 //or
 l.sfleu  rA,rB   //11100100101...
```

Listing 1: The Result Input Pattern about Bugzilla 51 from HW-SCM.

*5.3.2 Causality Inference in DarkRISC-V and RISCV-mini*For CWE-1234, our findings indicate that any opcode pattern can trigger this bug, meaning there are no restrictions on the input instruction. This aligns with our expectation as the signal debug can override the `io_dmem_wen` signal, allowing the user to access the register interface regardless of the input instruction. To confirm our findings, we constructed another HW-SCM model that only relied on the property `io_dmem_wen#0 == 1`. Typically, the signal `io_dmem_wen` is used to monitor write access to storage devices in RISC-V cores. Our HW-SCM further revealed that only an opcode equal to `0100011` can set the signal `io_dmem_wen`, which is the STORE opcode as expected.

6 Related Work

A comparison between Microscope and its related works is elaborated to highlight the advantages and novelty of the proposed method in this section.

Approach	(Avg)Time	Replayable	Traces generated
Coppelia [21]	252 s	yes	≥ 1
JasperGold [4]	0.10 s	no	1
Microscope	21.72 s	yes	≥ 1

Table 3: Comparison with existing work on or1200 testbench

6.1 Comparison with Existing Works

We present a comparison of our work with the commercial tool, JasperGold FPV [4], and the publicly available Coppelia [21], as depicted in Table 3. *"Average Time"* is calculated based on six Bugzilla test cases. *"Replayable"* refers to whether the generated traces can be restarted from the hardware reset state, which decides if consecutive instructions can be obtained. *"Traces Generated"* denotes the number of triggered traces produced given a certain assertion. The detailed definitions of *"Replayable"* and *"Traces Generated"* can also be found at [21]. Since Coppelia doesn't release detailed configuration for every test case, we use the best average time cost claims from the original paper. The JasperGold time cost is calculated by using JasperGold FPV run cover property counterexample generation time. With optimizations applied, Coppelia requires minutes to generate the exploit, whereas our method accomplishes in less than a minute. The improvement can be attributed to the fact that Microscope eliminates the need to convert Verilog designs into C++ representations,i.e., the constructed HW-SCM is directly built on the original design with precise timing information. Additionally, Coppelia cannot directly apply sequential assertions for generating software exploits. If the user wants to specify expected behavior over multiple clock cycles, they would have to manually insert extra flip-flops into the design and log the signal value from the previous clock cycles. Microscope can directly annotate signals from different layers of the HW-SCM to describe the sequential assertion since our model adds different timestamps for each signal.

In the domain of hardware-software boundary causal inference, there currently exists no dedicated commercial tool. However, some applications offer partial support for this functionality, albeit with a significant manual workload. Cadence JasperGold FPV, a cornerstone of the JasperGold Apps framework, can generate a counterexample for a user-defined assertion. Notably, JasperGold can utilize the cover property to infer input patterns and responds more swiftly than other tools. Nevertheless, a limitation is that only one counterexample is provided in each verification pass. To address all potential software-exploited bugs, a verification engineer must repeatedly execute the checking procedure until no counter traces are reported. This iterative process is both tedious and time-consuming. Furthermore, when the same baseline constraint is input into both JasperGold FPV and Microscope, there are instances where the exploits generated by JasperGold are not replayable. Specifically, the generated counterexample trace might commence from an intermediate state rather than the initial reset state. This is particularly evident for vulnerabilities activated by state transitions, i.e., those requiring a specific continuous input sequence to trigger the payload.

6.2 Relationship between Information Flow Tracking and Microscope

IFT-based approaches have been well developed to detect bugs in purely hardware RTL design [10]. When applied to the analysis of a hardware system, IFT aims to protect confidentiality and integrity by detecting sneaky paths of sensitive information leakage and modification. Causality inference is the process where causes are inferred from data. In the security area, it determines whether a security event is causally dependent on a preceding trigger event. The relationship between IFT and causality inference in terms of analyzing cyber attacks is first presented in [13] and elaborated in [14].

While IFT helps identify where a security breach may have occurred by monitoring data's movement, causality inference can provide the "how" and "why"—how the breach happened and why it occurred in that particular way. We, therefore, consider IFT as a fundamental infrastructure upon which we base our causality analysis. Specifically, IFT helps generate data-flow graph in Microscope. In this procedure, IFT identifies which signals are related to each other. This can potentially allow the SMT solver to more effectively partition the problem or apply heuristics, which could improve efficiency. However, it's important to note that even without using IFT as the infrastructure, the HW-SCM can still be obtained by directly parsing from RT-level (i.e., Verilog) codes. The trade-off, however, is that the time cost of SMT solving will be increased.

7 Conclusion and Future Work

This paper introduces the HW-SCM to apply the causality inference to RT-level hardware and software security co-verification. The proposed Microscope framework is developed to heuristically identify bug structures, and then automatically infer the potential malicious input instruction sequences that can trigger these bugs. Microscope is thoroughly validated using SoC-level platforms. In the future, we plan to collect open-source SoC platforms and insert CWE bugs so that more experiments can be carried out.

Acknowledgments

Portions of this work were supported by the National Science Foundation (CCF-2019310).

References

[1] 2022. Common Weakness Enumeration. https://cwe.mitre.org/.
[2] Baleegh Ahmad, Wei-Kai Liu, Luca Collini, Hammond Pearce, Jason M. Fung, Jonathan Valamehr, Mohammad Bidmeshki, Piotr Sapiecha, Steve Brown, Krishnendu Chakrabarty, Ramesh Karri, and Benjamin Tan. 2022. Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 157, 9 pages. https://doi.org/10.1145/3508352.3549369
[3] Cadar, Cristian and Dunbar, Daniel and Engler, Dawson R and others. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.
[4] Cadence. 2021. *JasperGold Platform and Formal Property Verification App User Guide.*
[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. 249–266.
[6] DarkLife. n.d.. *DarkRISCV: An open-source RISC-V processor.* https://github.com/darklife/darkriscv
[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[8] Nicole Fern. 2020. CWE-1262: Improper Access Control for Register Interface. (2020).
[9] Madelyn Glymour, Judea Pearl, and Nicholas P Jewell. 2016. *Causal inference in statistics: A primer*. John Wiley & Sons.
[10] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware Information Flow Tracking. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–39.
[11] Arun Kanuparthi, Hareesh Khattri, Parbati Kumar Manna, and Narasimha Kumar V Mangipudi. 2020. CWE-1234: Hardware Internal or Debug Modes Allow Override of Locks. (2020).
[12] Arun Kanuparthi, Hareesh Khattri, Parbati Kumar Manna, and Narasimha Kumar V Mangipudi. 2020. CWE-1245: Improper Finite State Machines (FSMs) in Hardware Logic. (2020).
[13] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 503–515.
[14] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. 2018. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation.. In *NDSS*, Vol. 2. 4.
[15] Openrisc. n.d.. *OR1200: An open-source implementation of the OpenRISC 1200 processor*. https://github.com/openrisc/or1200
[16] Sandip Ray, Eric Peeters, Mark M Tehranipoor, and Swarup Bhunia. 2017. System-on-chip platform security assurance: Architecture and validation. *Proc. IEEE* 106, 1 (2017), 21–37.
[17] Synopsys. 2019. *VC Formal Verification User Guide*. Version P-2019.06-SP2.
[18] Yunfeng Tao. 2009. An introduction to assertion-based verification. In *2009 IEEE 8th International Conference on ASIC*. IEEE, 1318–1323.
[19] ucb-bar. n.d.. RISCVmini: simple RISC-V 3-stage pipeline written in Chisel. https://github.com/ucb-bar/riscv-mini
[20] Clifford Wolf. 2016. Yosys open synthesis suite.
[21] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. 2018. End-to-end automated exploit generation for validating the security of processor designs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 815–827.