# A Temporal Causality Model for SoC-Scale Security Formal Verification at the Hardware-Software Boundary

Zhaoxiang Liu, Xiaolong Guo
*Kansas State University*
ECE Department
Manhattan, KS, US
zxliu@ksu.edu
guoxiaolong@ksu.edu

Orlando Arias
*University of Massachusetts Lowell*
ECE Department
Lowell, MA, US
orlando_arias@uml.edu

Dean Sullivan
*University of New Hampshire*
ECE Department
Durham, NH, US
dean.sullivan@unh.edu

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

***Abstract*—We propose Microscope, a new framework that addresses growing security issues in System-on-Chip (SoC) designs due to their complexity and involvement of third-party vendors. Traditional methods are inadequate for identifying software-exploited hardware vulnerabilities, and existing solutions for hardware-software co-verification often fall short. The framework has been proven effective through extensive testing on SoC benchmarks and it has outperformed existing methods and commercial tools in comparative analyses.**

***Keywords*—Causality Inference, Hardware Security, Hardware and Software Co-Verification, System-on-Chip**

## I. Introduction and Contribution

SoC designers face growing security concerns due to increased SoC complexity and the involvement of third-party vendors. The spotlight on hardware-software co-verification has grown due to the emergence of transient execution attacks [1]. Existing solutions such as Coppelia [2] convert hardware language to software for analysis, but this approach lacks precision and is inconvenient. Additionally, tools like Yosys [3], Cadence JasperGold [4], and Synopsys VC Formal [5] are not designed for co-verification, so while they may identify some vulnerabilities using assertion-based techniques, doubts remain about uncovering all interaction traces between software and hardware.

In response to these challenges, this paper presents and enhances Microscope [6], an innovative framework designed to infer potential software instruction patterns that expose hardware vulnerabilities. Specifically, we enhance the Structural Causal Model (SCM) [7] 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 2 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 SoC benchmarks and widely compared with existing methods.

## II. Methodology

This section delineates the HW-SCM as a multi-layer graph model that facilitates causality inference for automated hardware-software co-analysis.

### A. HW-SCM definition and Graph Example

HW-SCM extends the foundational concept of SCMs [7], 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:

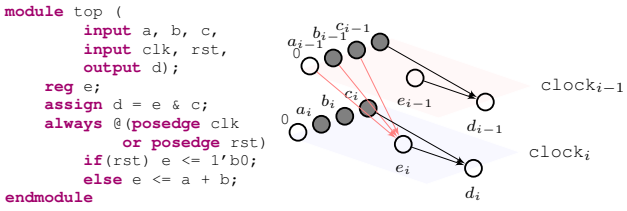
$$f_{comb} = \{f_i : X_i \to y_i \mid y_i \in HW_i\}, \tag{1}$$

$$f_{seq} = \{f_i : X_{i-1} \to y_i \mid y_i \in HW_i\}, \tag{2}$$

Here, $X_i \subseteq (SW_i \cup HW_i) \backslash \{y_i\}$ 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 $i-1$.

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. In this example, the output signal `d` depends on both signals `e` and `c`. The signal `c` is a direct input port, while signal `e` is updated by inputs `a` and `b` at the positive edge of the clock signal (`clk`). The graph model is depicted in Figure 1b, 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.

In Figure 1b, 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




(a) Verilog example. (b) Graph of HW-SCM.

Fig. 1: Graph based HW-SCM

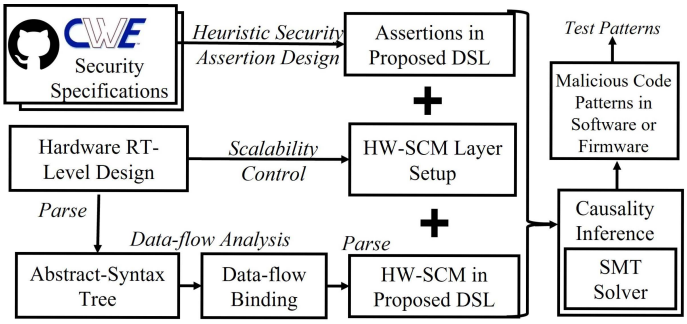



Fig. 2: Working procedure of the Microscope framework

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* 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.

### B. 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 written in SMT-LIB 2. A heuristic approach is adopted to design/obtain assertions using extant hardware databases like CWE [8] and Bugzilla [2] as references. Simultaneously, the number of layers is determined 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 SMT-LIB 2.

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 (1) and higher-layer sequential inputs as defined in Equation (2). Consequently, the inputs from each layer are accumulated and interpreted as instruction 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.

### C. 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 five types of design flaws extracted from the OR1200 commit history.

Specifically, 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 I and explained in the following paragraphs.

*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 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.

*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,

*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.

*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.

## III. Experiment Overview & Comparison

Our testing environment consisted of a machine running Ubuntu 20.04, equipped with an i9-12900K processor and 32GB of memory. Z3 [9] is applied as the SMT solver in this experiment. The experiments on or1200[10] processor and

| Benchmark | Vulnerability | Language | Cell Number | Layers | Time | Inference | Vulnerability Description |
|---|---|---|---|---|---|---|---|
| OR1200 | Bugzilla #51 | Verilog | 20,668 | 6 | 23.91 s | ✓ | **Comparison wrong for unsigned inequality with different MSB.** ALU module yeild incorrect result for unsigned comparation . |
| OR1200 | Bugzilla #76 | Verilog | 20,714 | 6 | 24.03 s | ✓ | |
| OR1200 | Bugzilla #88 | Verilog | 20,901 | 6 | 23.80 s | ✓ | **l.extw instructions behave incorrectly** No need to explicitly apply an extend operation when using the l.extw instruction. |
| OR1200 | Bugzilla #90 | Verilog | 20,743 | 6 | 18.44 s | ✓ | **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. |
| OR1200 | Bugzilla #97 | Verilog | 20,945 | 6 | 18.42 s | ✓ | **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. |

TABLE I: Experiment Result

| Approach | (Avg)Time | Replayable | Traces generated |
|---|---|---|---|
| Coppelia [2] | 252 s | yes | $\geq 1$ |
| JasperGold [4] | 0.10 s | no | 1 |
| Microscope | 21.72 s | yes | $\geq 1$ |

TABLE II: Experiment summary on or1200 processor

comparison with JasperGold FPV [4] and Coppelia [2] are summarized in Table II. *Average Time* is calculated based on six Bugzilla test cases(*#51,#76,#88,#90,#97*)[2]. *Replayable* refers to whether the generated traces can be restarted from the reset state. *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 [2].

Since Coppelia doesn't release detailed configuration for every test case, we use the best average time cost claims from the original paper. With optimizations applied, Coppelia requires minutes to generate the exploit, whereas our method accomplishes in less than a minute. 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.

JasperGold quickly infers input patterns using *cover property* counterexample generation but only provides one counterexample per pass. To address all potential software-exploited bugs, a verification engineer must repeatedly execute the tedious and time-consuming checking procedure until no counter traces are reported. When using the same baseline constraint in both JasperGold FPV and Microscope, JasperGold's exploit may start from an intermediate state, not the initial reset state, making it non-replayable. This is particularly evident for vulnerabilities activated by state transitions, i.e., those requiring a specific continuous input sequence to trigger the payload.

## IV. Related Work

Table III summarizes related methods and evaluates them in terms of types of bug detection, hardware overheads, timing behavior analysis, working stage, as well as scalability in an effort to differentiate Microscope.

The second column *SW Bugs* means if the method is capable of detecting a pure software level bug. In the listed approaches, only LDX and MCI are developed to check program execution. *HW Bugs* in the third column means detection of purely hardware bugs where bug payloads and triggers are all in the hardware. As an example, hardware Trojans from Trusthub [27] belong to this category. In particular, Coppelia and Fuzzing-based methods cannot detect purely hardware bugs. The column of *SW+HW Bugs* indicates hardware and software collaborative bugs, where transient execution is a typical example. As a result, IFT-based RTL analysis and commercial EDA tools do not consider software related bugs.

The *HW Overhead* indicates if extra hardware area is needed for deploying the method. Extra logic is added by language-based Caisson/Sapper and runtime DIFT approaches. The *Timing* column stands for if the method is able to deal with bugs related to clock-cycles in hardware. In this column, we assume all runtime and simulation based approaches, such as RTLIFT, can handle the sequential logic. As a result, static taint analysis methods SecVerilog and QIF-Verilog cannot process clock-cycles accurately. CWE Scanner focuses on checking states rather than handling cycles. The next column indicates if the method works at pre-silicon stage or post-silicon stage. The last column provides three scalability supporting levels – IP-level and SoC-level. In general, hardware simulation/fuzzing based methods and post-silicon approaches often have a better scalability performance.

As we can see from Table III, Microscope is developed to check software and hardware collaborative bugs. Compared

| Method | Bug Type | | | Logic Type | | Silicon | | HW Overhead | Scale |
|---|---|---|---|---|---|---|---|---|---|
| | SW | HW | SW&HW | Comb | Seq | Pre- | Post- | | |
| **Microscope (this work)** | ✗ | ✓ | ✓ | ● | ● | ● | – | – | SoC |
| LDX[11], MCI[12] | ✓ | ✗ | ✗ | ● | ● | – | ● | – | SoC |
| Caisson [13], Sapper [14] | ✗ | ✓ | ✗ | ● | ● | – | ● | ● | IP |
| SecVerilog [15], QIF-Verilog [16], RTLIFT[17] | ✗ | ✓ | ✗ | ● | ○ | ● | – | – | IP |
| CWE Scanner[18] | ✗ | ✓ | ✗ | ● | – | ● | – | – | SoC |
| SPV[19], FSV[5], Radix[20] | ✗ | ✓ | ✗ | ● | ● | ● | – | – | SoC |
| Coppelia [2] | ✗ | ✗ | ✓ | ● | ● | ● | – | – | SoC |
| BugsBunny[21], SpecDoctor[22] | ✗ | ✗ | ✓ | ● | ● | ● | – | – | SoC |
| Formal-HDL[23], VeriCoq-IFT[24] | ✗ | ✓ | ✓ | ● | ● | ● | – | – | IP |
| IP-Tag[25], CellIFT[26] | ✗ | ✓ | ✓ | ● | ● | – | ● | ● | SoC |

TABLE III: Microscope vs Existing methods

with Coppelia and Fuzzing-based methods, Microscope models the software and hardware from the hardware viewpoint so that it can detect hardware bugs also. Recent works such as SpecDoctor and Logic Fuzzer focus on transient executions or bugs inside the processor and memory system. Compared with them, Microscope is able to detect instruction patterns that trigger bugs in the peripheral IPs. Moreover, domain specific language developed in Formal-HDL and VeriCoq-IFT formalize software as a hardware representation, however, it utilizes an interactive platform to build the model. As an automatic framework, Microscope supports larger scale benchmarks than Formal-HDL and VeriCoq-IFT. As a pre-silicon method, Microscope works on the compilation stage and does not bring in extra hardware overhead. Further, Microscope is capable of handling clock-cycle events by introducing the timing window.

## V. Conclusion

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 both IP-level and SoC-level platforms. A major concern is the lack of well-maintained SoC-level security benchmarks for testing the security of formal verification methods. Therefore, in the future, we plan to collect the 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, First Award Program of ARISE in EPSCoR 2148878).

## References

[1] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 249–266.

[2] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "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, 2018, pp. 815–827.

[3] C. Wolf, "Yosys open synthesis suite," 2016.

[4] Cadence, *JasperGold Platform and Formal Property Verification App User Guide*, June 2021.

[5] Synopsys, *VC Formal Verification User Guide*, December 2019, Version P-2019.06-SP2.

[6] Z. Liu, K. Chen, D. Sullivan, O. Arias, R. Dutta, and X. Guo, "Microscope: Causal inference crossing the hardware and software boundary," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, (To Appear).

[7] M. Glymour, J. Pearl, and N. P. Jewell, *Causal inference in statistics: A primer*. John Wiley & Sons, 2016.

[8] "Common weakness enumeration," 2022, https://cwe.mitre.org/.

[9] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[10] "OR1200: An open-source implementation of the openrisc 1200 processor," https://github.com/openrisc/or1200.

[11] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "Ldx: Causality inference by lightweight dual execution," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 503–515.

[12] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Ciocarlie *et al.*, "Mci: Modeling-based causality inference in audit logging for attack investigation." in *NDSS*, vol. 2, 2018, p. 4.

[13] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 109–120.

[14] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 97–112.

[15] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[16] X. Guo, R. G. Dutta, J. He, M. Tehranipoor, and Y. Jin, "Qif-verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment," in *IEEE Symposium on Hardware Oriented Security and Trust (HOST)*, 2019.

[17] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1691–1696.

[18] B. Ahmad, W.-K. Liu, L. Collini, H. Pearce, J. M. Fung, J. Valamehr, M. Bidmeshki, P. Sapiecha, S. Brown, K. Chakrabarty *et al.*, "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*, 2022, pp. 1–9.

[19] Cadence, *Security Path Verification App User Guide*, December 2020.
[20] T. C. R.-S. offical website, "Radix from cycuity," 2022, https://cycuity.com/solutions/.
[21] H. Ragab, K. Koning, H. Bos, and C. Giuffrida, "Bugsbunny: Hopping to rtl targets with a directed hardware-design fuzzer."
[22] J. Hur, S. Song, S. Kim, and B. Lee, "Specdoctor: Differential fuzz testing to find transient execution vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1473–1487.
[23] X. Guo, R. G. Dutta, and Y. Jin, "Eliminating the hardware-software boundary: A proof-carrying approach for trust evaluation on computer systems," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 2, pp. 405–417, 2017.
[24] M.-M. Bidmeshki and Y. Makris, "Toward automatic proof generation for information flow policies in third-party hardware ip," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015, pp. 163–168.
[25] K. Chen, O. Arias, X. Guo, Q. Deng, and Y. Jin, "Ip-tag: Tag-based runtime 3pip hardware trojan detection in soc platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.
[26] F. Solt, B. Gras, and K. Razavi, "{CellIFT}: Leveraging cells for scalable and precise dynamic information flow tracking in {RTL}," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2549–2566.
[27] Trust-Hub, "Benchmarks," `https://www.trust-hub.org/`.