HUnTer: Hardware Underneath Trigger for Exploiting SoC-level Vulnerabilities

Sree Ranjani Rajendran, Shams Tarek, Benjamin M Hicks,
Hadi M Kamali, Farimah Farahmandi, Mark Tehranipoor
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA.
Email: {rajendrans, shams.tarek, benjamin.hicks, h.mardanikamali}@ufl.edu, {farimah, tehranipoor}@ece.ufl.edu

***Abstract*—Systems-on-chip (SoCs) have become increasingly large and complex, resulting in new threats and vulnerabilities, mainly related to system-level flaws. However, the system-level verification process, whose violation may lead to exploiting a hardware vulnerability, is not studied comprehensively due to the lack of decisive (security) requirements and properties from the SoC designer's perspective. To enable a more comprehensive verification for system-level properties, this paper presents HUnTer (Hardware Underneath Trigger), a framework for identifying sets (sequences) of instructions at the processor unit (PU) that unveils the underneath hardware vulnerabilities. The HUnTer framework automates (i) threat modeling, (ii) threat-based formal verification, (iii) generation of counterexamples, and (iv) generation of snippet code for exploiting the vulnerability. The HUnTer framework also defines a security coverage metric (*HUnT_Coverage*) to measure the performance and efficacy of the proposed approach. Using the HUnTer framework on a RISC-V-based open-source SoC architecture, we conduct a wide variety of case studies of Trust-HUB vulnerabilities to demonstrate the high effectiveness of the proposed framework.**

***Index Terms*—System-on-chip (SoC), Security Properties, Security Verification, SW-Exploitable Hardware Vulnerabilities.**

I. Introduction

The ever-increasing size and complexity of modern systems-on-chip (SoCs) have pushed many semiconductor manufacturers to follow a globalized (horizontal) integrated circuit (IC) supply chain with pressures for faster IC turnaround (tight time-to-market). SoC architectures are becoming increasingly complex as they integrate more and more cores, memory, 3^{rd}-party intellectual properties (3PIPs), etc. It significantly increases the risk that hardware-oriented security vulnerabilities remain unaddressed. So, the number of attacks targeting the modern complex SoCs is concerningly increasing [1]. In past recent years, we witnessed more and more sophisticated attacks combining the design abstraction layers from hardware to software for exploiting the vulnerabilities. For instance, attacks like Meltdown [2] and Spectre [3] show how an attacker can exploit bugs from the software level (or cross level) of abstraction to circumvent security checks and countermeasures for leaking the hardware-domain security asset, e.g., authentication data.

Over time, different research endeavors show that the origin of a significant portion of such vulnerabilities is at hardware, resulting in multiple re-spins or even huge revenue loss [4], [5]. The semiconductor industries try to incorporate the security development life (SDL) cycle [6] during the hardware development cycle. However, commercial electronic design automation (EDA) tools are more specialized for the functional aspect of verification, showing why a security-oriented verification-in-depth is a must for SoC verification, particularly for building testing cross-layer environments [7].

Since the industry-standard (functional-oriented) verification tools mostly fail for security-specific verification and detecting detect cross-layer attacks [4] in modern SoCs, thus recent studies focus on crafting specialized, either processor-centric or IP-centric, security verification, such as security-oriented assertion-based verification [8], information flow tracking (IFT) [9], symbolic/concolic testing [10], and more recently self-evolutionary testing (e.g., fuzz and penetration testing) [11]–[13]. Even though they have been successful to some extent, these techniques still lack the fundamentals that are necessary for SDL. The security-oriented assertion-based techniques require huge detailed expert knowledge from the design perspective to be accomplished. IFT-based techniques suffer from low scalability, particularly at gate-level netlist and on complex designs. Symbolic/Concolic testing are also compute-intensive approaches that may lead to failure due to non-scalability or runtime timeout. Techniques like fuzz testing heavily rely on coverage metrics that make vulnerability detection more observant-based. This shows that these methodologies are just shedding light on system-level security verification.

Given the processor's central importance to the modern SoCs, the base of vulnerability identification is recently shifted more towards the processor's higher-level language (HLL) and its instruction set architecture (ISA). For instance, in fuzz testing, the tuning (mutation) point has pushed into the software (HLL) running on the processor, and probing the internal signals leads to vulnerability detection [11], [14], [15]. It is also consistent with the fact that in larger and more complex SoCs, it is rarely the case that all internal IPs are directly accessible, which also pushes the adversary to sit up on the processor (higher-level of abstraction) level for exploiting the vulnerabilities and establishing new attacks.

In this paper, by relying on the above-mentioned level of abstraction for monitoring, we introduce HUnTer, which is a framework for automating the exploiting of hardware-oriented (IP-level) vulnerabilities from the processor perspective using formal methods. The HUnTer framework focuses on open-

978-3-9819263-7-8/DATE23/© 2023 EDAA

source RISC-V ISA, which has attracted the attention of industry and academia and is becoming the mainstream [16], and already integrated into open-source SoCs such as Rocket Chip (SiFive core), Ariane SoC, PULPino, PULPissimo, Shakti, Chipyard, and OPENPULP [17], [18]. The main contributions of this paper are as follows:

(1) We introduce HUnTer, a formal-based automation framework from threat modeling to (ISA-based) snippet code generation for software-exploitable hardware vulnerabilities.

(2) In HUnTer, we define two different notions: (i) HUnT_CEX, which is the automatically formal-based generated counterexample (CEX) generated for the exploited vulnerability, and (ii) HUnTer_Coverage, which is a coverage metric for monitoring use cases and possible vulnerabilities.

(3) By using these two notions, i.e., HUnT_CEX and HUnTer_Coverage, we validate the efficiency of HUnTer on RISC-V-based Ariane SoC using a wide variety of case studies of Trust-HUB vulnerabilities [19].

II. Background and Previous Work

Fig. 1 demonstrates the overview of a modern SoC that consists of multiple cores, memories, tens of IPs, and a wide variety of peripherals. In today's globalized IC supply chain, each one of these components may be provided by an independent 3^{rd}-party vendor. In such a scenario, each component may have few/many security assets, i.e., any data whose leakage can lead to catastrophic consequences with significant revenue loss. Per each asset, a set of requirements will be defined, known as security properties that must be met to avoid known security issues. Additionally, unknown issues are always a threat to the SoC, and it is unclear how to be caught [12], [20]. It compelled the semiconductor manufacturers to employ security verification as an integral part of the SoC implementation cycle, which was previously accomplished using conventional manual code review testing techniques, simulation, and emulation [7]. However, these techniques are time-consuming, non-scalable, and expert-knowledge-dependent, making them impractical, particularly when it comes to interpretation at different levels of abstraction, from software to hardware. Additionally, in the last few years, we witnessed a surge-style increasing trend of identified/investigated vulnerabilities by CWE [21]. This is the main incentive of newer security verification techniques, whose main aims are to provide (1) *scalability*, (2) *performance*, (3) *high coverage*, (4) *adaptability*, and (5) *being EDA-friendly*. The following are the mainstream SoC security verification:

(I) Formal-based Security Verification: In this case, the verification team specifies how security properties must be implemented (e.g., via SystemVerilog assertions) for a potential vulnerability (property-based checking) [7], [22]. By defining the security properties (w.r.t. the security assets), formal tools, e.g., Cadence JasperGold, will verify the expected security behavior.

(II) Symbolic/Concolic Execution: In this breed of solutions, assertions (that are defined based on security assets and properties) must be crafted into the design, using a set of tags,

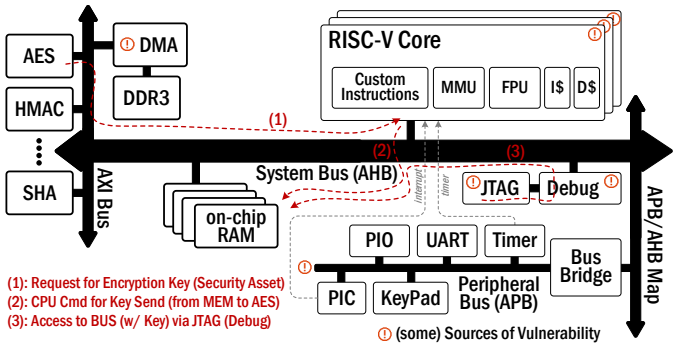

Fig. 1: Typical RISC-V-based SoC with a Simple Vulnerability Test Case.

symbols, and breakpoints, and after a set of structural/functional graph/path specification, test patterns will be generated for to push the assertions condition to be met [10], [23].

(III) Information Flow Tracking: In IFT [9], [24], the verification engine is built on a label-based propagation, in which, based on the policy/property, a set of input signals will be labeled and based on the propagation of the labels, detection of vulnerabilities will be accomplished.

(IV) Fuzz Testing: Fuzz testing is an evolutionary mechanism with the capability of self-refinement for hardware vulnerability detection [11], [14], [15]. All fuzzing techniques on hardware utilize coverage-based testing, and they mutate the inputs from the software level to the hardware for exploiting the vulnerabilities.

Solutions of groups (I), (II), and (III) are mostly at the RTL level of abstraction, and input/output is more IP-centric. However, solutions of the group (IV) shifted the test environment mostly to the software level of abstraction, and the test cases are mostly HLL executing on the processor(s) of the SoC(s). In [4], it is represented that detecting (and eliminating) security-relevant hardware vulnerabilities at the RTL level will face fundamental challenges: (i) Cross-modular (and cross-layer) vulnerabilities (effects of inter-IP or inter-layer communication); (ii) timing side-channel vulnerabilities (different timing flow for different input/output pairs); (iii) non-register states-related vulnerabilities (cache-based). Amongst the existing security verification solutions (groups I-IV), fuzz testing has attracted significant attention very recently as it targets the SoC vulnerabilities from the software level.

Table I compares the existing methodologies for SoC secu-

TABLE I: Comparison of Security Verification Technqiues in SoC.

Method	Model	Limit	Scale	Automate	Abstract
Formal [8], [22]	whitebox	No support for Cross-layer & cross-modular verification	high	low	RTL
Concolic [10]	whitebox (manual)	Path Explosion, False Positive	low	moderate	RTL
Symbolic [23]	whitebox (manual)	Only processor-level	low	moderate	RTL
IFT [9], [24]	whitebox	Path Explosion, State space Explosion	very low	high	RTL/HLL
Fuzzing [11], [14], [15]	graybox	Reliance on coverage metrics	moderate	high	RTL/HLL

rity verification. Compared to these techniques, our proposed HUnTer tries to address the challenges discussed above and simultaneously benefits from the formal methods for generating code snippets at the abstraction level of the software for exploiting the hardware-oriented security vulnerabilities. It allows us also to cover cross-layer (different abstracts) and cross-modular (high scalability) vulnerabilities. Our experiments show that one code snippet can trigger multiple vulnerabilities in the hardware layer of the processor. This multi-target snippet generation significantly reduces the effort of verification by time and complexity.

A. Threat Model

In light of the main objective of the *HUnTer* framework, which is to construct a software-exploitable hardware vulnerability detection system, we define the following as the main entry points of threats: (i) an unauthorized application that is capable of being executed on the processor core of the system (in our case it would be the execution of unauthorized software on RISC-V core of the Ariane SoC). With complete access to user space, unauthorized software can execute illegal instructions, functions, and system calls in user mode. (ii) malicious-inserted hardware integrated into the platform and may have access to some IPs and peripherals. Considering these sources, The main goal of the adversary here is to exploit vulnerabilities at the software level to bypass security regulations, access security-critical data, and illegal actions.

III. HARDWARE UNDERNEATH TRIGGER (HUNTER)

The main aim of the proposed HUnTer framework is twofold: (1) building a framework that can be engaged for exploiting the vulnerabilities (hardware vulnerability underneath the processor) using software-level test cases, and (2) Engaging formal tools for generating the test cases at software level of abstraction. Considering that each application (software) running on the CPU core of the SoC will be translated to machine-language (assembly) code based on the ISA, the hardware accomplishes the execution of the machine code (with the preferred sequence). The main aim of the HUnTer framework is to benefit from software-to-machine and machine-to-hardware translation to provoke vulnerabilities in the hardware domain. For instance, a vulnerability in the hardware may cause an interruption in the sequence of instructions during the processing cycle, which may affect the system's confidentiality, reliability, and integrity and leaves a backdoor to other software/hardware attacks. The HUnTer focuses on such coupling between abstraction layers and generates snippet codes to exploit the vulnerabilities. HUnTer uses the potential hardware weaknesses listed in CWE [21] to automation threat modeling, counterexample (HUnT_CEX) generation, and constructing the code snippets based on HUnT_CEXs that trigger the underneath hardware vulnerability in the SoC. Fig. 2 demonstrates the major steps of the HUnTer framework, whose descriptions are as follows:

Step ①: Identifying and Selecting Threat Model: The main focus of the HUnTer framework is on the most important

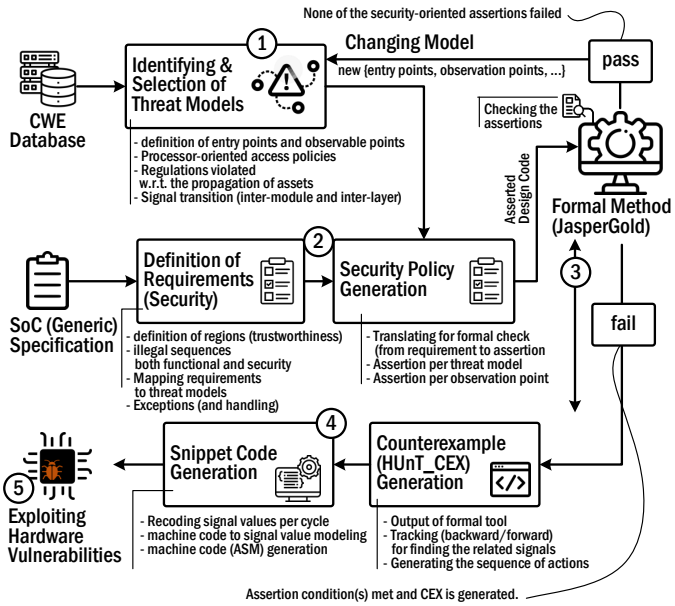

Fig. 2: The Major Steps of the Proposed HUnTer framework

CWE vulnerabilities and their associated threat models. For the sake of simplicity, we adopted 6 SoC vulnerability benchmark designs from Trust-Hub [19] as test case scenarios in this paper, and we implemented all in the Ariane core of RISC-V [25]. There are a number of different traits that are used to characterize these vulnerabilities, such as their security objectives, requirements, and implications, all listed in Table II. It is worth mentioning that all of these vulnerabilities fall under the CAPEC attack pattern of *privilege escalation* [26][1]. In the benchmarks, *control & status register* (CSR) is considered the attack surface and modified to incorporate the privilege escalation-related vulnerabilities. For instance, one of the vulnerabilities includes the privilege level issue during inter-processor interrupt handling. Interrupt handling may allow the host processor to access the security-critical information of the target processor if the host processor is at a higher privilege level than the target processor.

Step ②: Security Property Generation: To realize the security properties in this study, we consider Ariane SoC, which is equipped with a 6-state, in-order RISC-V processor [25]. After determining the threat model (Step ①), the security properties are developed based on the SoC's design specification and security requirements. An instance of one property has been demonstrated in Listing 1. This specific property is used to formally verify the security requirement of *the Ariane core processor to accept debug requests only when the processor is at the highest privilege level*, meaning that the RISC-V processor must be in the machine mode while accepting the debug request. All the security properties are developed based on such kind of security requirements of the design. For this specific study, we have developed 28 security properties to formally verify the Ariane core, along with the *15* cover prop-

[1]The Privilege escalation is when an adversary can perform unauthorized access to important registers and fuses from a lower privilege level.

TABLE II: Attack Pattern and Threat Model of the SoC Vulnerability Database Considered in the Proposed HUnTer Framework

SoC benchmark	Category ID	Weakness Name	Vulnerability	Security Objective	Security Requirement	Attack Pattern & Description	Ramification
SoC-V1	CWE-1198	Privilege Separation and Access Control Issues	Unauthorized access to registers	Confidentiality and integrity	Correct privilege level should be assigned and maintained	CAPEC-233: Privilege Escalation	Access control
SoC-V2	CWE-266	Incorrect Privilege Assignment	Improper handling and assignment of privileges				Illegal interrupt
SoC-V3	CWE-280	Improper Handling of Insufficient Permissions or Privileges	Improper assignment or handling of permissions to registers				
SoC-V4	CWE-1272	Erroneous Update of Read and Write Enable signal	Unauthorized Enable Signal Assertion during Privilege Violation				
SoC-V5	CWE-1262	Unauthorized Page Access Request	Unauthorized Page Access Request				
SoC-V6	CWE-1262	Illegal PMP access after mismatch	Illegal PMP access after mismatch				

erties to verify the scenarios of illegal instruction interruption at consequent clock cycle (will be discussed in Section IV).

```
@(posedge clk_i) (debug_reg_i == 1'b1)
   ariane.csr_regfile_i.priv_lvl_o == RISCV::PRIV_LVL_M)
```

Listing 1: An Instance of Security Property in Ariance SoC.

Step ③: Formal Verification and HUnT_CEX Generation: After the generation of all security properties and considering the threat model(s), the HUnTer framework invokes the formal tool (i.e., Cadence JasperGold). In formal verification, the objective is to detect any security violations inside the design (based on security properties). If the design *passes* formal verification, HUnTer will choose another threat model and will return to ①. This continues until a CEX (HUnT_CEX) is obtained for a property violation. In this study, since the threat model follows a privilege escalation threat model, the CEX will definitely show a scenario where an adversary can gain unauthorized access to any lower privilege level. So, if the design *fails* formal verification, the HUnTer framework will check the design scenarios and conquer a sequence of instructions for each vulnerability at the hardware level. Throughout our experiments, we find that the set of instructions from the CEX is unique for a scenario that violates a security property. Table III shows the output of JasperGold for the CSR module in the Ariane SoC. The table covers what security assets are considered, how many vulnerabilities are exploited, and whether HUnT_CEX is generated for the exploited vulnerability or not.

Step ④: Automatic Snippet Code Extraction: One of the main steps of the HUnTer framework is the automation of snippet code extraction based on HUnT_CEX(s). The snippet code generator is the script that takes the counterexample (HUnT_CEX) from the formal tool (Cadence JasperGold followed by sequencing by the HUnTer) as its input. Then, based on the HUnT_CEX (sequence of interactions), the HUnTer records the signal values at each clock cycle. For instance, the signal values of the program counter (a.k.a. instruction pointer) and 32-bit machine code for each clock cycle are collected at the decode stage of the Ariane RISC-V core. From the 32-bit decoded machine code, the field values of register addresses, operands, and functions can be obtained by HUnTer.

```
.section .text
.globl start
_start:
add       x0,     x1,     x2
sw        t0,     40(t1)
sw        t0,     40(t1)
loop1     addi    x5,     x1,     1
beq       x8,     x0,     loop1
add       x0,     x0,     0
jalr      ra,     0(x0)
add       x0,     x0,     0
add       x0,     x0,     0
sw        t0,     40(t1)
sw        t0,     40(t1)
sw        t0,     40(t1)
sw        t0,     40(t1)
csrrci    x3,     0x7B0,  15
sw        t0,     40(t1)
sw        t0,     40(t1)
ecall
ecall
ecall
add       x0,     x0,     0
csrrci    x3,     0x7B0,  15
end:
```

Listing 2: Example Snippet Code for Asset #2 of Table III.

Further, the program counter and machine code represent a single instruction in the assembly that, when compiled and executed, would generate the HUnT_CEX. An intermediate file is generated with the 64-bit program counter (address)

Algorithm 1 Snippet Code Extraction

Ensure: $compile(ASM) \rightarrow CEX$

1: $ASM \leftarrow \{\}$ ▷ /*ASM is an empty .asm file*/
2: $binaries \leftarrow CEX.id\_stage\_i.instruction$
3: $addresses \leftarrow CEX.id\_stage\_i.decoded\_instruction.pc^{\{*1\}}$
4: **for** i in 0 to addresses.length-1 **do**
5: $\quad binary \leftarrow binaries[i]$
6: $\quad address \leftarrow addresses[i]$
7: $\quad instruction \leftarrow disassemble(address, binary)^{\{*2\}}$
8: $\quad ASM \leftarrow ASM + instruction$

$\{*1\}$: addresses[i] contains the instruction binary, binaries[i], for i in 0 to addresses.length-1

$\{*2\}$: disassemble(address, binary) returns the instruction string (e.g. add x0,x0,0). disassemble() is defined by the open-source RISC-V disassembler tool

TABLE III: JasperGold Formal security Verification Results of CSR module in SoC benchmarks

CWE Weakness & Module	Asset	Security Requirement	No.of Vul.	HUnT_CEX
Privilege Escalation or Violation & Control and Status Register file	Asset #1: Sensitive asset of the interrupt target	During inter-processor interrupt handling, the core should be in machine mode	1	✓
	Asset #2: Content of control and status register/ control and status bits of CSR	No invalid read or write should take place for control and status registers	2	✓
	Asset #3: Information regarding current instruction	If privilege is violated, the micro-architectural state or read/write should not be updated to CSR	1	✓
	Asset #4: Information regarding current instruction, Intermediate result of ongoing encryption operation	While returning from debugging, the previous privilege level should be restored	2	✓
	Asset #5: Registers	When privilege level/mode changes, CSRs should be flushed before re-fetching the next instruction	2	✓
	Asset #6: Read access exception signal in CSR	During debug mode, no exception request should be taken	2	✓
	Asset #7: Protected memory location	Memory page access should not be given to users with improper privilege level	2	✓
	Asset #8: Address translation information	In case of physical memory entry mismatch with virtual memory, memory access should be given only in Machine mode	1	✓

followed by a 32-bit machine code (data) for every clock cycle in the HUnT_CEX. The intermediate file is passed to an open-source RISC-V disassembler, which outputs the instruction represented by the address and data pair. The sequence of instructions that creates a CEX for the considered threat model is obtained as the assembly language (ASM) and is considered a snippet code. Algorithm 1 describes the snippet code extraction algorithm developed in the HUnTer framework. By developing this algorithm in HUnTer and employing it for asset #2 of Table III, a snippet code will be generated (see Listing 2).

Step ⑤: Exploiting Vulnerability in RISC-V toolchain: The HUnTer framework benefits from the RISC-V GNU compiler toolchain [27] to exploit the vulnerabilities revealed and observed by snippet codes in Ariane RISC-V tool-chain. This step serves as a verification (confirmation) stage by running the snippet codes in the RISC-V toolchain environment with the Spike RISC-V ISA simulator. After running, the log file is dumped to observe the contents of each register in the CSR module. Listing 3 is an instance of log file after running the snippet code for asset #2 of Table III. Obviously, in this code running at user space with no privilege, CSR-related operations (`csrw` and `csrr`) are unauthorized. *If there is any invalid read/write happening to any control and status register, an adversary can get access to an address register that may hold critical information of the processor*. In Listing 2, it is observed that read access is enabled to the register `SR_mtvec` since the hardware weakness of privilege violation is exploited as the vulnerability by running a code snippet in the simulator of the RISC-V environment. It is also clear that the registers `CSR_scratch` and `CSR_mhartid` have read/write options which are the additional vulnerabilities exploited by the snippet code.

To see how snippet code generation is effective for exploiting the vulnerabilities, we define a security coverage metric, called *HUnTer_Coverage*. *HUnTer_Coverage* provides an empirical valuation of the total number of snippet codes to exploit the total number of vulnerabilities with the number of assets considered for security verification. In fact, *HUnTer_Coverage* demonstrates that, per each asset, and per each CEX generated for a violated security property, whether a snippet code is generated or not. The performance of HUnTer is validated using the *HUnTer_Coverage* metric in Section IV.

```
: core   0:
0x0000000080000268 (0x00000f93) li      t6, 0
0x000000008000026c (0x34001073) csrw    mscratch, zero
0x0000000080000270 (0x00000297) auipc   t0, 0x0
0x0000000080000274 (0xd9428293) addi    t0, t0, -620
0x0000000080000278 (0x30529073) csrw    mtvec, t0
0x000000008000027c (0x30502373) csrr    t1, mtvec
0x0000000080000280 (0x00629063) bne     t0, t1, pc + 0
0x0000000080000284 (0x00010117) auipc   sp, 0x10
0x0000000080000288 (0xc3c10113) addi    sp, sp, -964
0x000000008000028c (0xf14026f3) csrr    a3, mhartid
0x0000000080000290 (0x00c69613) slli    a2, a3, 12
```

Listing 3: Vulnerability Trigger Log (SoC-V1 for Asset #2 of Table III).

$$HUnTer\_Coverage = \frac{Total\,No.\,of\,Snippet\,Code}{(No.\,of\,Asset \times No.\,of\,HUnT\_CEX)} \times 100 \quad (1)$$

IV. Experimental Results and Evaluation

To evaluate the efficiency of the HUnTer framework, as demonstrated in Table II, a set of Trust-Hub vulnerabilities has been targeted. The vulnerabilities are induced in Ariane core's CSR [25] and tested using the Ariane SoC test harness framework. In this model, the privilege level violation will be a gateway for unauthorized access to the registers and fuses. Similar to other formal-based techniques, formal verification is carried out using the Cadence JasperGold tool [8]. All steps of the HUnTer are established using the RISC-V platform, including the toolchain, RISC-V ISA Spike simulator environment, and the proxy kernel (pk) environment.

Table III shows the results of vulnerability detection, the number of them, and the capability of reporting HUnT_CEX by the framework. For this experiment, we focused on privilege escalation as the main threat model on the CSR module of the RISC-V core in the Ariane SoC. As shown, the HUnTer was able to generate HUnT_CEX for all cases. As shown, the number of assets considered for evaluation is 8. For these assets, 13 security properties out of 28 are violated,

TABLE IV: HUnTer_Coverage for Different Security Vulnerabilities through *fifteen* Different Scenarios (1-15 Clock Cycle(s)).

SoC Benchmark	HUnTer_Coverage (%)															Average
	S_1^*	S_2^*	S_3^*	S_4^*	S_5^*	S_6^*	S_7^*	S_8^*	S_9^*	S_{10}^*	S_{11}^*	S_{12}^*	S_{13}^*	S_{14}^*	S_{15}^*	
SoC-V1	83.65	84.62	85.58	85.58	83.65	89.42	83.65	87.50	87.50	87.50	83.65	80.77	86.54	81.73	89.42	85.38
SoC-V2	80.77	79.81	82.69	79.81	80.77	84.62	79.81	86.54	81.73	81.73	89.42	80.77	85.58	86.54	84.62	83.01
SoC-V3	84.62	78.85	79.81	87.50	83.65	84.62	89.42	89.42	79.81	90.38	84.62	78.85	79.81	84.62	84.62	84.04
SoC-V4	85.58	79.81	84.62	81.73	91.35	82.69	85.58	85.58	90.38	84.62	82.69	75.96	91.35	83.65	87.50	84.87
SoC-V5	79.81	81.73	82.69	80.77	84.62	87.50	75.96	89.42	80.77	90.38	91.35	86.54	79.81	89.42	83.65	84.29
SoC-V6	87.50	84.62	91.35	81.73	80.77	82.69	84.62	87.50	91.35	90.38	92.31	84.62	89.42	91.35	88.46	87.24
															Average	84.41

S_i^*: Scenario used to validate the HUnTer performance, where, S^*i is the scenario i to verify the occurrence of illegal instruction for i consecutive clock cycle(s)

whose HUnT_CEX are generated as the output of the formal verification tool. Apart from the security properties, coverage-based properties are written to evaluate the sequence of events. This allows us in HUnTer to capture the sequence and impact of instructions in ISA. The assumption is that the interruption of illegal instruction in the ISA flow will exploit the hardware weakness to provide access control to the registers in the CSR module. For this, we consider 15 different scenarios, each for verifying the sequence of instruction for consecutive clock cycles. Violation of this sequence results in generating new HUnT_CEX. Table IV reflects the HUnTer_coverage for all targeted vulnerabilities (from Table II), which is based on Eq. 1. Per each vulnerability, as mentioned previously, 15 different scenarios are selected each for different clock cycles. Clock cycle counts are swept (1-15) to build different snippet codes per scenario. On average, the HUnter framework can achieve +84% coverage, which means that for 8 assets with 13 different vulnerabilities (8×13 = 104 combinations), more than $84\%\times104$ = 87 are exploited with available snippet code.

V. Conclusion and Future Work

This paper proposes HUnTer, a formal-based SoC security verification that focuses on uncovering the vulnerability of the underneath hardware through the integrated processor unit. HunTer correlates the exploited vulnerabilities found by the formal method with snippet codes generated based on ISA. By doing this, per each counterexample (HUnT_CEX) generated by the formal tool, the HUnTer creates a snippet code for exploiting the vulnerabilities from the software level of abstraction. HUnTer has been evaluated through numerous SoC benchmarks with induced vulnerabilities from the Trust-Hub repository. By concentrating on the privilege escalation threat model and CSR-based assets, HUnTer shows more than 84% coverage for generating snippet codes in ISA. In future work, HUnTer focuses on both weakness and asset-based vulnerabilities to fork out the SoCs confidentiality/integrity.

References

[1] M. Tehranipoor, *Emerging Topics in Hardware Security*. Springer, 2021.

[2] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[3] M. Lipp *et al.*, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[4] G. Dessouky *et al.*, "Hardfails: Insights into software-exploitable hardware bugs," in *USENIX Security Symposium*, 2019, pp. 213–230.

[5] Z. Kenjar *et al.*, "V0ltpwn: Attacking x86 processor integrity from software," in *USENIX Security Symposium*, 2020, pp. 1445–1461.

[6] M. Howard *et al.*, *The security development lifecycle*. Microsoft Press Redmond, 2006, vol. 8.

[7] F. Farahmandi *et al.*, *System-on-Chip Security*. Springer, 2020.

[8] N. Farzana *et al.*, "Soc security verification using property checking," in *IEEE International Test Conference (ITC)*, 2019, pp. 1–10.

[9] A. Ardeshiricham *et al.*, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 1691–1696.

[10] X. Meng *et al.*, "Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 466–477, 2021.

[11] K. Laeufer *et al.*, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[12] K. Z. Azar *et al.*, "Fuzz, penetration, and ai testing for soc security verification: Challenges and solutions," *Cryptology ePrint Archive*, 2022.

[13] H. Al-Shaikh *et al.*, "Sharpen: Soc security verification by hardware penetration test," in *Asian and South Pacific Conference on Design Automation Conference (ASP-DAC)*, 2023, pp. 1–6.

[14] T. Trippel *et al.*, "Fuzzing hardware like software," in *USENIX Security Symposium*, 2022, pp. 3237–3254.

[15] A. Tyagi *et al.*, "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," *arXiv preprint arXiv:2201.09941*, 2022.

[16] K. Asanovic *et al.*, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[17] A. Traber *et al.*, "Pulpino: A small single-core risc-v soc," in *3rd RISCV Workshop*, 2016.

[18] K. Asanovic *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016.

[19] Trust-Hub, "Trust-hub Repository," https://trust-hub.org/.

[20] S. Bhunia *et al.*, *Hardware security: a hands-on learning approach*. Morgan Kaufmann, 2018.

[21] MITRE, "HW CWEs," https://cwe.mitre.org/data/definitions/1194.html.

[22] P. Bhamidipati *et al.*, "Security analysis of a system-on-chip using assertion-based verification," in *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2021, pp. 826–831.

[23] A. Ahmed *et al.*, "Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution," in *2018 IEEE International Test Conference (ITC)*, 2018, pp. 1–10.

[24] W. Hu *et al.*, "Hardware information flow tracking," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–39, 2021.

[25] F. Zaruba *et al.*, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[26] S. Barnum, "Common attack pattern enumeration and classification (capec) schema," *Department of Homeland Security*, 2008.

[27] RISC-V Software Collaboration, "RISC-V GNU Compiler Toolchain," https://github.com/riscv-collab/riscv-gnu-toolchain.