Hardware Security Benchmarks for Open-Source SystemVerilog Designs

Jayden Rogers*, Niyaz Shakeel†, Xiao Tan†, Samantha Espinosa†,
Divya Mankani†, Cade Chabra†, Kaki Ryan†, Cynthia Sturton†
*North Carolina A&T *jsrogers2@aggies.ncat.edu*
†University of North Carolina at Chapel Hill
{*niyaz, kakiryan, csturton*}*@cs.unc.edu,*
{*tanxiao, sespi, divyakm, cade.chabra*}*@unc.edu*

***Abstract*—The need to verify the security of hardware designs has led to new property-based verification and property-generation methods. Supporting this research are the open-source designs and security-critical properties for development and evaluation. While many open-source designs exist, there remains a need for more properties.**

This paper addresses that need. We develop 120 SystemVerilog Assertions for four open-source designs. We validate the properties against the associated designs and develop metrics with which to assess the benchmarks. The properties are open-source and have been accepted for inclusion in the Trust-Hub database.

***Index Terms*—Properties, SystemVerilog Assertions, Register Transfer Level, Verification, Formal Methods, Hardware Security**

I. Introduction

"For better or worse, benchmarks shape a field" (David Patterson, 2018 [50]). Formally verifying the security of hardware designs is an active area of research producing new tools [55], [42], [3], [46] and uncovering new security bugs and vulnerabilities [69], [42], [3]. To support this research, the hardware security community has developed and published a large number of open-source hardware designs useful for tool development (e.g., [48], [58], [68]), along with buggy designs useful for tool evaluation (e.g., [27], [65]).

Unfortunately, sets of security properties are not similarly available. Many verification methods are property-based: the verification engine takes a (possibly buggy) hardware design and a set of properties as input and outputs any found counterexamples (CEX) to the properties. There is general consensus that developing properties is a laborious task [66], [39], with ongoing research into generating properties automatically [70], [18], [71], most recently using large language models [49], [21], [38], yet few resources provide publicly available properties tied to a particular snapshot of a design. This work is a step toward that goal.

Our contribution is pragmatic: we provide four sets of security properties for widely studied buggy designs: the OR1200 OpenRISC CPU core [48], the PULPissimo SoC with RISC-V core [58], and two versions of the OpenPiton SoC with the CVA6 RISC-V core [68], [5]. OR1200 is snapshotted with inserted bugs from prior work [69], [33], [55]; PULPissimo and OpenPiton use Hack@DAC versions from 2018, 2019, and 2021 [28], [30], [29], [32]. All properties are written as SystemVerilog Assertions (SVA), the industry standard for trace property specification [1], and the security bugs they cover are categorized according to the Common Weakness Enumeration (CWE) taxonomy [45]. We provide this categorization for OR1200 and supplement the existing categorization for the Hack@DAC designs.

We developed 71 properties for the OR1200, detecting all known bugs and offering additional security assurance over prior work. For PULPissimo, we create 20 properties to address 31 known bugs, an improvement over prior efforts that reported (but did not publish) 15 properties. For the 2019 and 2021 OpenPiton SoCs, we created 11 and 20 properties for 66 and 99 bugs, respectively. Using Cadence JasperGold, the industry-standard formal verification tool, we validated that the properties catch the bugs. The properties and design snapshots are released as benchmarks for future research.

The properties derive from three sources: prior security analyses; Hack@DAC security bugs developed with industry, and our own analyses of widely used open-source designs. Together, they capture a snapshot of the state of the field of hardware security verification. By assessing the coverage of the set, we can gain insights into where the field has focused its attention and where there is room for further analysis. We assess their CWE coverage, with emphasis on the *Most Important Hardware Weaknesses* identified by the CWE website.

Finally, we present a case study illustrating how missing properties hinder reproducibility. Our hope is that our properties will accelerate future security verification research and provide a baseline for evaluating automated property generation techniques.

This paper makes three contributions.

- A set of well-vetted SVA properties for four buggy designs and their snapshotted designs: https://github.com/HWSec-UNC/verification-benchmarks
- An evaluation of property coverage against CWEs, emphasizing *Most Important Hardware Weaknesses*.
- A case study demonstrating the reproducibility challenges when properties are missing from the public record.

II. Background

We provide context for our work and discuss related work.

A. Scope

We focus on property-based formal verification for hardware security validation. Functional validation is out of scope, though the boundary between security properties and functional properties can be blurry, as security flaws often stem from functional bugs. Our interest is in properties that identify vulnerabilities exploitable by attackers. These resources may or may not be appropriate for functional validation activities.

B. Threat Model

Our properties detect bugs in the RTL design that can lead to security vulnerabilities. We consider an attacker with the ability to execute code of their choosing on the hardware post-deployment. This can be achieved, for example, by hosting a malicious website containing Javascript that is downloaded and run on the target machine when the victim visits the site. The attacker's code may or may not execute with elevated privilege. These properties apply to design-time security verification. Malicious modifications made post-verification, attacks during fabrication or in the supply chain, and physical attacks on the machine post-deployment will not be detected by these properties.

C. Available Open-Source Resources

Trust-Hub. TrustHub is an NSF-funded website designed to support research and education in hardware security [65]. The website serves as a directory for various hardware security resources and also offers resources directly, including Trojan-infected design components, taxonomies, and datasets related to physical hardware security. Two resources on the Trust-Hub site are the Security Property/Rule Database and the SOC Vulnerability Database.

The Security Property/Rule Database [24] provides RTL and gate-level modules with accompanying SVA properties. While useful, it is limited: only a handful of properties exist for components of the CV32E40P RISC-V core, and most entries are small components (e.g., write-once registers, FIFOs, SPI, RSA, AES), each with one to five properties. The SoC Vulnerability Database [59] lists six vulnerabilities in the CVA6 RISC-V CPU with one to four SVA properties each. However, the version of the CVA6 design for which the properties were written is not mentioned, and the properties may require significant changes to be applicable for the current version of the design.

Our work expands on these two databases at a larger scale, providing 71, 20, 11, and 20 properties, respectively, for four buggy CPU and SoC designs, along with snapshotted versions of the designs for which the properties were written. Our repository has been accepted for inclusion in the Trust-Hub directory, making it easy for the community to find the resource.

Hack@DAC Competitions. The Hack@DAC competitions [57], [27] have been a boon to the hardware security community. In the annual competition, competitors are given the RTL of an SoC design and must find the inserted security bugs. The organizers are a team of academic researchers and industry practitioners, and the bugs inserted reflect real-world vulnerabilities.

The design used in the 2018 instance of the competition is a PULPissimo SoC [28] and has become a standard benchmark for hardware security research (e.g., [55], [38], [46]), thanks in part to the organizers providing natural-language descriptions of the bugs inserted into the design [15].

Recently, the OpenPiton SoC designs used in the 2019 and 2021 competitions were made available as well [30], [32], along with the bug descriptions for the 2019 version [29] and an overview of the 2021 version [12].

However, the available resources stop short of pinpointing the actual bugs in the RTL. Research teams using these designs as a benchmark for the development or evaluation of their tool must first establish ground truth by finding the bug in the RTL through whatever means possible. The SVA properties we provide will boost the usefulness of these benchmark designs by pinpointing the bug in the design and acting as a baseline set of assertions for use in future research.

CWE Database. The Common Weakness Enumeration (CWE) database from MITRE is a community-maintained resource that categorizes common flaws in software and hardware that often lead to vulnerabilities [45]. Entries in the database describe the flaw, provide information about how a vulnerability may manifest, and may include example code snippets. The database can be used as a guide when writing the formally stated properties needed for security verification [25], [54], [14]. We used CWEs as a guide when writing assertions and link to the appropriate CWE for the inserted bugs when they were missing.

D. Related Work

Property Generation. A variety of techniques have been used to automatically generate security properties for hardware designs. The most recent papers explore using large language model (LLM) code generation to generate RTL properties [49], [38], [43], [22], [74], [22]. Other techniques include cross-design property translation, mining properties from design behavior, and using CWE descriptions to generate properties [70], [71], [21], [17], [18], [20], [16], [51], [19].

While some papers provide a handful of the properties generated [49], [17], [21], [18], [38], [20], [16], we found only one paper that provides the complete set of generated properties [71]. The automated approaches typically generate thousands of properties; selecting the most relevant ones is done either manually or through statistical analysis. Both approaches benefit from having a set of known-good properties as a starting point.

Property-Based Formal Verification. Many new analyses and frameworks for formally verifying security properties of a design have been developed [42], [69], [73], [72], [33], [62],

[55], [41], [23], [4], [40], [56]. However, many papers do not report the properties used in the evaluation. A few do [33], [60], [36], and a few more report one or two of the properties used [3], [53], [52]. But if we consider only papers using the industry-standard SVA for property specification, we are left with few properties [53], [52], [60], making comparisons between tools difficult. The properties we provide will make it easier for new research in security verification techniques to make comparisons with the existing state of the art.

Fuzzing. Many recent studies in hardware security develop fuzzing tools [64], [8], [66], [11], [67], [10], [9], [26], [63], [35], [34]. Fuzzing is a simulation-based method like testing, but fuzzing benchmarks do not include expected outputs. Instead, fuzzers detect bugs using a golden reference model, security properties, or both. Our SVA properties may be suitable for some fuzzers, though many modern fuzzers rely on golden models. While this avoids the need for security properties, it has its own limitations.

Taint Tracking. Taint tracking is a simulation-based method that has seen academic [61], [44] and commercial [13] success in the hardware security community [37]. These tools instrument the design to trace information flow during simulation and typically use their own specification languages for properties. Our SVA properties are not directly compatible. Although not a formal verification method, taint tracking also requires well written properties, and there have been efforts toward easing the property writing process [54], [17].

III. Property Development

We develop SVA trace properties for four processor and SoC designs. The properties are safety properties: properties are violated by reaching an undesirable state or finite sequence of events. The full set of properties and design snapshots are available at: https://github.com/HWSec-UNC/verification-benchmarks, organized by the design with support for community contribution.

A. Designs Studied

Properties are tightly coupled to a design and often need to be rewritten when the design is updated due to changes in timing, signal names, or data flow between registers, even when high-level behavior stays the same. To ensure reproducibility, our benchmarks include the exact commit version and a static snapshot for each design.

1) OR1200: The OR1200 is a widely studied 5-stage pipelined single-core processor [48], [33], [69], [55]. Security and functional bugs found in the design over time have been documented in the public OpenCores [47], Bugzilla [7] and Mitre CWE databases [45]. In addition, bugs are documented as issues opened by developers on their GitHub repository [48]. Of note, there are cases where security researchers have identified native bugs in one generation of the processor that a subsequent version or commit claims to have resolved [70], but the bugs reappear or persist over time [69].

Our benchmark snapshots the OR1200 in a buggy state containing 31 security bugs collected from prior work [33], [70] along with a set of 71 security properties targeting desired secure behavior and the inserted bugs. Fifteen of these SVA properties are inspired by OVL properties provided in prior work [33]; the remaining properties are based on properties described (but not provided) in prior work [6], [70], [71]. Despite prior use of OR1200, the community still lacks an available set of SVA properties written for the design and a buggy benchmark that can serve as ground truth during evaluation. This benchmark fills that gap.

2) Hack@DAC 2018: The PULPissimo SoC is a RISC-V-based SoC with a 6-stage pipeline and security enhancements. The version used in the Hack@DAC 2018 competition [28] and studied in the HardFails paper [15] contains both native and inserted bugs.

We developed a set of properties targeting the bugs known to be present in the version used in the Hack@DAC competition. In writing these properties, we used the bug descriptions available in the literature [28], [15], [42], and map each bug to a corresponding CWE. Our repository includes a snapshot of the buggy design, the full set of SVA properties, and CWE annotations to support reproducible evaluation.

3) Hack@DAC 2019 and 2021: The designs used for the HACK@DAC 2019 [30] and 2021 [31] competitions were based on the CVA6 SoC (formerly known as the Ariane SoC). These two designs have 66 and 99 bugs inserted, respectively. Some of these vulnerabilities were native to the design and some, similar to HACK@DAC 2018, were inserted by the competition organizers.

Using bug descriptions and case studies published in the literature, we develop properties to find the bugs. To validate the assertions in JasperGold, we had to resolve a series of syntax errors and make small fixes to compile the entire SoC. The repository provides these finalized versions with 11 and 20 properties for each design, respectively.

B. Property Development

Developing properties requires understanding the design and its security-critical assets, studying known vulnerabilities, and translating bug descriptions written in natural language to formal assertions. The process is iterative; properties are tested and refined through simulation and formal verification. The amount of time required to write one property depends on the writer's level of expertise in SVA specifications and knowledge of the design. In our experience, the time ranged from a couple of hours to one week. Each property is one or two lines of code and adds under 0.10 seconds to verification time in Cadence JasperGold.

The properties are manually written with one of two goals:

1) Capture desired secure behavior of a design; or
2) Capture specific known buggy behavior of a design.

In the first case, the goal is to formally specify secure behavior. Violations are not expected unless the design is flawed. These properties are developed using the instruction set architecture (ISA) specification and available design documentation. In the second case, the goal is to expose known vulnerabilities; violations are expected. Ideally, a complete specification meeting the first goal would be sufficient to capture all security flaws. In practice, it is helpful to work toward both goals in tandem.

Table I summarizes the provenance of the security properties evaluated across the four hardware designs. We categorized each property based on its origin: derived from the ISA specification, written to target a known native bug in the design, created to detect a manually inserted bug, or translated from a property written for another design. The OR1200 properties were primarily derived from the ISA (41) and augmented with a smaller number of inserted bug checks (29) and one translated property. In contrast, the Hack@DAC designs relied more heavily on inserted bug properties, with 21, 11, and 13 properties for the 2018, 2019, and 2021 designs, respectively. The 2018 and 2021 sets also include a handful of native or translated properties, while the 2019 set was exclusively constructed to target inserted bugs.

Design	ISA	Native Bug	Inserted Bug	Translated Property
OR1200	41	0	29	1
Hack@DAC 2018	0	10	21	0
Hack@DAC 2019	0	0	11	0
Hack@DAC 2021	4	1	13	2

TABLE I: Property provenance for each design.

C. *Property Examples and Challenges*

Challenge: Bug descriptions provide limited information. Writing assertions from bug descriptions requires expert-level familiarity with the codebase. For example, Hack@DAC 2021 Bug 76 is described as "some of the register lock registers are not locked by register locks" [32]. Our investigation led us to the code shown in Listing 1, with the apparent bug on Line 5. When writing to register 2, if locked, it copies `reglk_mem[3]` instead of keeping `reglk_mem[2]`. We found confirmation of this bug from a description of a similar bug by Ahmad et al. in their work on CWE Analysis [2].

Listing 1: The buggy line of code `reglk_wrapper` module of Hack@DAC 2021 OpenPiton SoC

```
 1 else if(en && we)
 2  case(address[7:3])
 3    0: reglk_mem[0] <= reglk_ctrl[3] ? reglk_mem[0]:wdata;
 4    1: reglk_mem[1] <= reglk_ctrl[1] ? reglk_mem[1]:wdata;
 5    2: reglk_mem[2] <= reglk_ctrl[1] ? reglk_mem[3]:wdata;
 6    3: reglk_mem[3] <= reglk_ctrl[1] ? reglk_mem[3]:wdata;
 7    4: reglk_mem[4] <= reglk_ctrl[1] ? reglk_mem[4]:wdata;
 8    5: reglk_mem[5] <= reglk_ctrl[1] ? reglk_mem[5]:wdata;
 9    default: ;
10  endcase
```

Decoding arbitrarily complex FSM states and transitions was often required. For example, one bug in the 2021 design has the description: "Able to write using JTAG without password." This suggests a vulnerability in the JTAG module, but required reverse-engineering a complex FSM to understand. As shown in Listing 3, the FSM can transition from IDLE to WRITE without checking `pass_check == 1`. This reverse-engineering process can take many hours for a large design.

Similarly, many bugs require detailed knowledge of the peripheral interactions with the SoC's memory interface. Peripherals may be mapped to fixed memory indices per configuration. One Hack@DAC 2019 bug states: "Processor access to CLINT (core level interrupt controller) grants it access to PLIC (processor level interrupt controller) regardless of PLIC configuration." Resolving this required identifying the hardcoded indices in the RTL in Listing 2, where PLIC maps to 6 and CLINT to 7.

Listing 2: The buggy line of code `axi_node_intf_wrap` module of Hack@DAC 2019 OpenPiton SoC

```
for (i=0; i<NB_SUBORDINATE; i++)
begin
for (j=0; j<NB_MANAGER; j++)
 begin
  assign connectivity_map_o[i][j] =
           access_ctrl_i[i][j][priv_lvl_i] ||
           ((j==6) && access_ctrl_i[i][7][priv_lvl_i]);
 end
end
```

Challenge: Properties have limited reusability. The above examples highlight how tightly coupled assertions are to the designs they are written for. As a result, similar bug descriptions across design generations rarely translated to reusable properties, as signal names, modules, and RTL structure often diverged significantly. For example, the same PLIC/CLINT bug is present in the 2021 design, but writing a property for it was challenging. The RTL was updated, and despite being able to identify peripheral mappings, the relevant logic was hard to locate.

Challenge: Bug descriptions can be misleading. Bug descriptions may point to a module or area of the design, but not to the specific logic, and simple keyword searches may point to irrelevant modules or signals. Simulation environments with waveform viewers and signal tracing ease the process, but analysis remains complex, especially over multiple modules.

In addition, bug descriptions may not always provide clarity about the level of abstraction where the vulnerability exists. Some issues appear in low-level system/C code included in SoC repositories and fall outside the scope of RTL analysis. While prior work [2] flags some Hack@DAC 2019 and 2021 CWEs as out of scope, this isn't always obvious from the descriptions. For example, bug 4 (CWE 1191) and bug 25 (CWE 1268) both require functional simulation to reproduce. In other words, the vulnerabilities are only realizable at runtime, making property writing more challenging.

IV. Case Study: Examining the Role Properties Play in Reproducibility

The buggy PULPissimo SoC from the 2018 Hack@DAC competition [28] has become a standard benchmark for hardware security verification [42], [46], [55], [55], [60], [38]. This is largely due to the 2019 HardFails paper which documents its vulnerabilities and provides a public snapshot of the design [15]. However, to our knowledge, there is no public dataset of formal properties for this design despite widespread

Listing 3: Buggy FSM in JTAG module of 2021 OpenPiton SoC

```
case (state_q)
 Idle: begin
  // make sure that no error is sticky
  if(dmi_access && update_dr &&
      (error_q == DMINoError)) begin
   // save address and value
   address_d = dmi.address;
   data_d = dmi.data;
   if((dm::dtm_op_e'(dmi.op) == dm::DTM_READ) &&
       (pass_check | ~we_flag == 1))
    begin
     state_d = Read;
    end
    else if((dm::dtm_op_e'(dmi.op) == dm::DTM_WRITE) &&
             (pass_check == 1))
    begin
     state_d = Write;
    end
    else if(dm::dtm_op_e'(dmi.op) == dm::DTM_PASS)
    begin
     state_d = Write;
     pass_mode = 1'b1;
    end
    // else this is a nop and we can stay here
  end
 end
```

use. Some works include one or two properties [46], [49], or natural-language description [42].

The issue is that there is no community expectation to provide SystemVerilog Assertions or equivalent RTL-level properties for identifying security flaws. As a result, every research group must independently write their own properties to use the benchmark. Not only is this time-consuming, but two groups independently writing properties to capture a bug are unlikely to write the same property, making it difficult to compare tool results.

To demonstrate this reproducibility challenge, we wrote SVA properties for each bug described in HardFails [15] and evaluated them using Cadence JasperGold's Formal Property Verification or *FPV*. Our evaluation uses the same PULPissimo SoC snapshot as HardFails. Two examples of assertions are shown in Listings 4 and 5.

Listing 4: Assertion for Bug 3 (Processor assigns privilege level of execution incorrectly from CSR.) in Hack@DAC 2018

```
assert(~((riscv_core.cs_registers_i.priv_lvl_n ==
     riscv_core.cs_registers_i.PRIV_LVL_M) && riscv_core.
     cs_registers_i.mstatus_n.mpp == riscv_core.
     cs_registers_i.PRIV_LVL_U)))
```

Listing 5: Assertion for Bug 13 (Faulty decoder state machine logic in RISC-V core results in a hang.) in Hack@DAC 2018

```
assert(riscv_controller.id_stage_i.controller_i.ctrl_fsm_ns
     == riscv_controller.id_stage_i.controller_i.DECODE)
     |=> (riscv_controller.id_stage_i.controller_i.
     ctrl_fsm_ns != riscv_controller.id_stage_i.controller_i
     .DECODE))
```

Table II shows the results of our reproducibility study. We group each bug by the CWE and evaluated whether our hand-written SVA properties could detect the flaws using JasperGold FPV. We compare our success in finding bugs to the corresponding JasperGold FPV results in the HardFails paper.

Despite using the same design and verification engine, we find bugs that HardFails only identified through manual inspection or didn't find at all. Conversely, they reported finding some bugs with JasperGold that we failed to write a property for. These discrepancies likely stem from differences in the properties used, underscoring the importance of open-sourcing formally stated assertions to support reproducibility.

CWE ID	No. of Bugs Found		Total Bugs Present
	Our Results	HardFails Results	
1203	3	3	3
1207	2	2	2
1206	1	1	1
1257	0	1	1
20	1	1	1
1221	1	1	1
1298	1	1	1
1329	2	1	3
1245	3	3	3
1247	1	1	1
1419	1	0	1
1271	0	0	1
1240	1	0	3
1220	1	0	5
325	1	0	1
1262	0	0	3

TABLE II: Comparing our success in writing assertions to find known bugs in the Hack@DAC2018 design to HardFails [15].

V. Analysis of Covered CWEs

The Common Weakness Enumeration (CWE), maintained by MITRE, standardizes descriptions of common security flaws in hardware and software. Each CWE captures a specific weakness, such as a bug that allows attackers to access protected memory or change important values in a device.

CWEs are hierarchical, with broad *parents*, and more specific *members* or *children* groupings. CWEs that share the same parent are called *siblings*. Abstract categories (e.g., CWE-1198) are discouraged in favor of specific children.

To help hardware designers focus on the most important issues, MITRE also released a list of the *The 2021 CWE Most Important Hardware Weaknesses* along with five additional CWEs that are *Weaknesses on the Cusp* that may become more common or serious. This list is curated with help from industry experts and researchers and is updated over time. It is *unranked*, meaning the CWEs are not ordered by how dangerous or common they are.

Table III provides an overview of how these CWEs relate to our dataset, listed in numerical order. We check whether a CWE is:

- Directly represented (i.e., appears in our dataset);
- Has a member that appears in our dataset;
- Has a parent that appears in our dataset; or
- Has a sibling represented.

This helps assess how well our findings align with known critical hardware issues. All of the Most Important Hardware

CWE	Brief Description	Appears in Dataset			
		This	Member	Parent	Sibling
CWE-1189	Improper Isolation of Shared Resources on SoC		✓		
CWE-1191	Improper Debug/Test Access Control		✓	✓	✓
CWE-1231	Improper Prevention of Lock Bit Modification		✓	✓	✓
CWE-1233	Missing Lock Bit Protection		✓	✓	✓
CWE-1240	Insecure Cryptographic Implementation	✓			
CWE-1244	Internal Asset Exposed to Debug Level	✓			
CWE-1256	Improper Restriction of Software Interfaces		✓		
CWE-1260	Overlapping Protected Memory Ranges	✓			
CWE-1272	Info Uncleared Before Debug/Power		✓	✓	✓
CWE-1274	Access Control for Volatile Memory		✓	✓	✓
CWE-1277	Firmware Not Updateable			✓	✓
CWE-1300	Protection of Physical Side Channels		✓		
CWE-226	Info Not Removed Before Reuse	✓			
CWE-1247	Protection Against Glitches	✓			
CWE-1262	Access Control for Register Interface	✓			
CWE-1331	Improper Isolation of Resources				✓
CWE-1332	Improper Handling of Faults		✓		✓

TABLE III: 2021 CWE Most Important Hardware Weaknesses and Related CWEs in Our Dataset

Weaknesses and the five Cusp entries are represented in our dataset in some form. For example, CWEs such as CWE-1240, CWE-1244, and CWE-1260, examples of the Most Important CWEs, are directly observed in our data. Others like CWE-1191 and CWE-1274 are indirectly represented through hierarchical relationships.

Our repository includes charts that provide a hierarchical, visual representation of the CWEs covered and how properties for each design relate to each other. Figure 1 shows a snippet of the chart for the Hack@DAC'21 SoC. The root node represents the full SoC design, with immediate children corresponding to the hardware modules of interest. There are six modules represented in this figure. Each module connects to properties associated with observed bugs, and these properties are linked via dotted arrows to CWEs describing the nature of the underlying vulnerabilities.

This visualization is helpful for teasing out observations and relationships present in our dataset. For example, we see some CWE reuse across modules. CWE-226 appears both in `aes0_wrapper` and `rsa_wrapper`. CWE-226 is related to secret data, so its recurrence in cryptographic modules is unsurprising and suggests opportunities for similar property development in other cryptographic blocks, both in this SoC design and across other designs.

CWE-1245 is represented in both `dmi_jtag` and `dma`. The CWE is related to improper finite state machine logic. While functionally distinct, both DMA and JTAG modules often rely on FSMs, hinting that this CWE could plausibly affect other control-heavy modules like UARTs, for example.

The `dmi_jtag` module stands out with multiple properties and two associated CWEs, flagging it as a potential security hotspot or a logically complex module. In contrast, there are modules (`aes_192`, `dma`, `sha256_wrapper`) that are each only associated with one property and one CWE. This may indicate more narrowly scoped issues, or a need for further analysis.

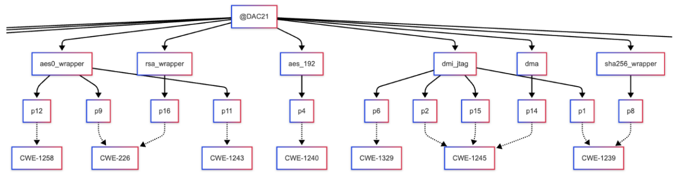

Fig. 1: Visualizing relationships between modules, properties, and CWEs.

VI. Discussion & Conclusion

In this paper, we address a critical gap in the hardware security community: the lack of open-source properties for verification. We find that natural language descriptions are insufficient records for both bugs and properties, and call for stronger community expectations that researchers publish the SVA or equivalent RT-level properties used in their work. We further stress the importance of bug reporting to include the buggy lines of code along with comments describing the context and behavior of the bug.

Through a case study of the Hack@DAC 2018 design, we demonstrate the importance of properties for the reproducibility of evaluation results. To help close this gap, we provide a set of SVA properties for four commonly used CPU and SoC designs as part of an open-source hardware verification benchmarks repository. Each benchmark includes properties mapped to known or inserted security flaws when applicable, along with documentation of the corresponding CWEs. By making these properties available, we aim to accelerate tool development, enable fair evaluation comparisons, and promote reproducible hardware security research.

VII. Acknowledgments

Thank you to the anonymous reviewers for their suggestions for improving the paper. This paper reports on work supported by the National Science Foundation under Grant No. CNS-2247754 and by Intel under the Scalable Assurance program.

References

[1] "IEEE standard for SystemVerilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[2] B. Ahmad, W. Liu, L. Collini, H. Pearce, J. M. Fung, J. Valamehr, M. Bidmeshki, P. Sapiecha, S. Brown, K. Chakrabarty, R. Karri, and B. Tan, "Don't CWEAT it: Toward CWE analysis techniques in early stages of hardware design," in *ICCAD*, ser. ICCAD '22. ACM, Oct. 2022.

[3] A. L. D. Antón, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "Fault attacks on access control in processors: Threat, formal analysis and microarchitectural mitigation," *IEEE Access*, 2023.

[4] 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*, 2017, pp. 1691–1696.

[5] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "OpenPiton: An open source manycore research framework," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232.

[6] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security Checkers: Detecting processor malicious inclusions at runtime," in *HOST*, 2011.

[7] Bugzilla, "Bugzilla." [Online]. Available: https://www.bugzilla.org/

[8] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," in *DAC*. IEEE, 2021, pp. 529–534.

[9] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi, "Processorfuzz: Processor fuzzing with control and status registers guidance," in *HOST*. IEEE, 2023, pp. 1–12.

[10] C. Chen, V. Gohil, R. Kande, A.-R. Sadeghi, and J. Rajendran, "Psofuzz: Fuzzing processors with particle swarm optimization," in *ICCAD*. IEEE, 2023, pp. 1–9.

[11] C. Chen, R. Kande, N. Nyugen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "Hypfuzz: Formal-assisted processor fuzzing," *arXiv preprint arXiv:2304.02485*, 2023.

[12] C. Chen, R. Kande, P. Mahmoody, A.-R. Sadeghi, and J. Rajendran, "Trusting the trust anchor: towards detecting cross-layer vulnerabilities with hardware fuzzing," in *DAC*, 2022, pp. 1379–1383.

[13] Cycuity. [Online]. Available: https://cycuity.com/

[14] Cycuity, "Radix coverage for hardware common weakness enumeration (CWE) guide." [Online]. Available: https://cycuity.com/type/white_paper/radix-coverage-for-hardware-common-weakness-enumeration-cwe-guide/

[15] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable Hardware Bugs," in *USENIX Security*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 213–230.

[16] C. Deutschbein, "Mining secure behavior of hardware designs," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2021.

[17] C. Deutschbein, A. Meza, F. Restuccia, R. Kastner, and C. Sturton, "Isadora: Automated information flow property generation for hardware designs," in *ASHES*. ACM, 2021.

[18] C. Deutschbein and C. Sturton, "Evaluating security specification mining for a CISC architecture," in *HOST*. IEEE, 2020.

[19] C. Deutschbein, A. Meza, F. Restuccia, M. Gregoire, R. Kastner, and C. Sturton, "Toward hardware security property generation at scale," *IEEE Security & Privacy*, vol. 20, no. 3, pp. 43–51, 2022.

[20] C. Deutschbein and C. Sturton, "Mining security critical linear temporal logic specifications for processors," in *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 2018, pp. 18–23.

[21] N. F. Dipu, A. Ayalasomayajula, M. Tehranipoor, and F. Farahmandi, "Agile: Automated assertion generation to detect information leakage vulnerabilities," *IEEE Transactions on Information Forensics and Security*, 2023.

[22] W. Fang, M. Li, M. Li, Z. Yan, S. Liu, Z. Xie, and H. Zhang, "Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms," 2024.

[23] W. Fang and H. Zhang, "Wasim: A word-level abstract symbolic simulation framework for hardware formal verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2023, pp. 11–18.

[24] N. Farzana, F. Rahman, M. Tehranipoor, and F. Farahmandi, "Soc security verification using property checking," in *ITC*, 2019, pp. 1–10.

[25] S. Gogri, P. Joshi, P. Vurikiti, N. Fern, M. Quinn, and J. Valamehr, "Texas A&M Hackin' Aggies' security verification strategies for the 2019 hack@dac competition," *IEEE Design & Test*, vol. 38, no. 1, pp. 30–38, 2021.

[26] V. Gohil, R. Kande, C. Chen, A.-R. Sadeghi, and J. Rajendran, "Mabfuzz: Multi-armed bandit algorithms for fuzzing processors," *arXiv preprint arXiv:2311.14594*, 2023.

[27] Hack@DAC, https://www.dac.com/Conference/HackDAC.

[28] Hack@DAC 2018 Phase 2 Buggy SoC. [Online]. Available: https://github.com/hackdac/hackdac_2018_beta

[29] Hack@DAC 2019. [Online]. Available: https://hackthesilicon.com/dac19-setup/

[30] Hack@DAC 2019 Alpha Stage SoC. [Online]. Available: https://github.com/HACK-EVENT/hackatdac19

[31] Hack@DAC 2021 Alpha Stage SoC. [Online]. Available: https://github.com/HACK-EVENT/hackatdac21

[32] Hack@DAC 2021 SoC. [Online]. Available: https://github.com/HACK-EVENT/hackatdac21

[33] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs," in *ASPLOS*, 2015, pp. 517–529.

[34] M. M. Hossain, N. F. Dipu, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "Taintfuzzer: Soc security verification using taint inference-enabled fuzzing," in *ICCAD*. IEEE, 2023, pp. 1–9.

[35] M. M. Hossain, A. Vafaei, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "Socfuzzer: Soc vulnerability detection using cost function enabled fuzz testing," in *DATE*. IEEE, 2023, pp. 1–6.

[36] W. Hu, A. Ardeshiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property specific information flow analysis for hardware security verification," in *ICCAD*, 2018, pp. 1–8.

[37] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware information flow tracking," *CSUR*, vol. 54, no. 4, pp. 1–39, 2021.

[38] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "Llm-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.

[39] R. Kastner, F. Restuccia, A. Meza, S. Ray, J. Fung, and C. Sturton, "Automating hardware security property generation," in *DAC*, 2022, pp. 1384–1387.

[40] Y. Lyu and P. Mishra, "Automated test generation for activation of assertions in RTL models," in *ASP-DAC*. IEEE, 2020, pp. 223–228.

[41] X. Meng, "Ensuring hardware robustness via security verification," Ph.D. dissertation, 2023.

[42] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu, "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.

[43] X. Meng, A. Srivastava, A. Arunachalam, A. Ray, P. H. Silva, R. Psiakis, Y. Makris, and K. Basu, "Unlocking hardware security assurance: The potential of llms," *arXiv preprint arXiv:2308.11042*, 2023.

[44] A. Meza, F. Restuccia, J. Oberg, D. Rizzo, and R. Kastner, "Security verification of the opentitan hardware root of trust," *IEEE S&P*, vol. 21, no. 3, pp. 27–36, 2023.

[45] MITRE, "Common Weakness Enumeration." [Online]. Available: https://cwe.mitre.org/

[46] J. Müller, M. R. Fadiheh, A. L. D. Antón, T. Eisenbarth, D. Stoffel, and W. Kunz, "A formal approach to confidentiality verification in socs at the register transfer level," in *DAC*. IEEE, 2021, pp. 991–996.

[47] OpenCores, "OpenCores." [Online]. Available: https://opencores.org/

[48] OR1200: OpenRISC 1200 Implementation. [Online]. Available: https://github.com/openrisc/or1200

[49] S. Paria, A. Dasgupta, and S. Bhunia, "Divas: An llm-based end-to-end framework for soc security analysis and policy-based protection," *arXiv preprint arXiv:2308.06932*, 2023.

[50] D. Patterson, "For better or worse, benchmarks shape a field: technical perspective," *Commun. ACM*, vol. 55, no. 7, p. 104, Jul. 2012.

[51] M. Qin, J. Li, J. Yan, Z. Hao, W. Hu, and B. Liu, "Ht-pgfv: Security-aware hardware trojan security property generation and formal security verification scheme," *Electronics*, vol. 13, no. 21, 2024.

[52] S. R. Rajendran, N. F. Dipu, S. Tarek, H. M. Kamali, F. Farahmandi, and M. Tehranipoor, "Exploring the abyss? unveiling systems-on-chip hardware vulnerabilities beneath software," *IEEE Transactions on Information Forensics and Security*, 2024.

[53] S. R. Rajendran, S. Tarek, B. M. Hicks, H. M. Kamali, F. Farahmandi, and M. Tehranipoor, "Hunter: Hardware underneath trigger for exploiting soc-level vulnerabilities," in *DATE*. IEEE, 2023, pp. 1–6.

[54] F. Restuccia, A. Meza, and R. Kastner, "AKER: A design and verification framework for safe andsecure SoC access control," 2021.

[55] K. Ryan and C. Sturton, "Sylvia: Countering the path explosion problem in the symbolic execution of hardware designs," in *FMCAD*. IEEE, 2023, pp. 110–121.

[56] ——, "Sylq-sv: Scaling symbolic execution of hardware designs with query caching," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 195–211. [Online]. Available: https://doi.org/10.1145/3676642.3736123

[57] A.-R. Sadeghi, J. Rajendran, and R. Kande, "Organizing the world's largest hardware security competition: Challenges, opportunities, and lessons learned," ser. GLSVLSI '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 95–100.

[58] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: an ultra-low-power pulpissimo soc in 22nm fdx," in *S3S*, 2018, pp. 1–3.

[59] SoC Vulnerability Database. [Online]. Available: http://cad4security.org/index.php/riscv-vulnerability-details/

[60] B. Solem, "Applying unique program execution checking in the development flow of industrial iot devices to prevent vulnerabilities for side-channel attacks," Master's thesis, NTNU, 2023.

[61] F. Solt, B. Gras, and K. Razavi, "{CellIFT}: Leveraging cells for scalable and precise dynamic information flow tracking in {RTL}," in *USENIX Security 22*, 2022, pp. 2549–2566.

[62] C. Sturton, M. Hicks, S. T. King, and J. M. Smith, "Finalfilter: Asserting security properties of a processor at runtime," *IEEE Micro*, vol. 39, no. 4, pp. 35–42, 2019.

[63] Y. Sugiyama, R. Matsuo, and R. Shioya, "Surgefuzz: Surge-aware directed fuzzing for cpu designs," in *ICCAD*. IEEE, 2023, pp. 1–9.

[64] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *USENIX Security 22*, 2022, pp. 3237–3254.

[65] Trust-Hub. [Online]. Available: https://www.trust-hub.org

[66] A. Tyagi, A. Crump, A.-R. Sadeghi, G. Persyn, J. Rajendran, P. Jauernig, and R. Kande, "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," *arXiv preprint arXiv:2201.09941*, 2022.

[67] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "{MorFuzz}: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *USENIX Security 23*, 2023, pp. 1307–1324.

[68] F. Zaruba and L. Benini, "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," *VLSI*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[69] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for validating the security of processor designs," in *MICRO*. IEEE, 2018, pp. 815–827.

[70] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying Security Critical Properties for the Dynamic Verification of a Processor," in *ASPLOS*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, p. 541–554.

[71] R. Zhang and C. Sturton, "Transys: Leveraging common security properties across hardware designs," in *IEEE S&P*. IEEE, 2020.

[72] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for processor security validation," *IEEE Design & Test*, vol. 38, no. 3, pp. 22–30, 2021.

[73] R. Zhang and C. Sturton, "A recursive strategy for symbolic execution to find exploits in hardware designs," in *Proceedings of the 2018 ACM SIGPLAN International Workshop on Formal Methods and Security*, ser. FMS 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–9. [Online]. Available: https://doi.org/10.1145/3219763.3219764

[74] Z. Zhang, G. Chadwick, H. McNally, Y. Zhao, and R. Mullins, "Llm4dv: Using large language models for hardware test stimuli generation," 2023.