Assert-O: Context-based Assertion Optimization using LLMs

Samit S. Miftah
samit.miftah@utdallas.edu
University of Texas at Dallas
Richardson, Texas, USA

Amisha Srivastava
amisha.srivastava@utdallas.edu
University of Texas at Dallas
Richardson, Texas, USA

Hyunmin Kim
hyunmin.kim@tii.ae
Technology Innovation Institute (TII)
Abu Dhabi, UAE

Kanad Basu
kanad.basu@utdallas.edu
University of Texas at Dallas
Richardson, Texas, USA

ABSTRACT

Modern computing relies on System-on-Chips (SoCs), integrating IP cores for complex functions. However, this integration introduces vulnerabilities, necessitating rigorous hardware security validation. The effectiveness of this validation depends on the security properties embedded in the SoC. Recent studies explore large language models (LLMs) for generating security properties, but these may not be directly optimized for validation. Manual intervention remains necessary to reduce their number. Security validation methods that rely on human expertise are not scalable as they are time-intensive and prone to human error. In order to address these issues, we introduce Assert-O, an automated framework designed to derive security properties from SoC documentation and optimize the generated properties. It also ranks the properties based on the security vulnerabilities they are associated with, thereby streamlining the validation process. Our method leverages hardware documentation to initially create security properties, which are subsequently consolidated and prioritized based on their level of criticality. This approach serves to expedite the validation procedure. Assert-O is trained on documentation of six IPs from OpenTitan. To evaluate our proposed method, Assert-O was assessed on five other modules from OpenTitan. Assert-O was able to generate 183 properties, which was further optimized to reduce them to 138 properties. Subsequently, these properties were ranked based on their impact on the security of the overall system.

KEYWORDS

Large Language Models, Hardware Verification, Hardware Security

ACM Reference Format:
Samit S. Miftah, Amisha Srivastava, Hyunmin Kim, and Kanad Basu. 2024. Assert-O: Context-based Assertion Optimization using LLMs. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24), June 12–14, 2024, Clearwater, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649476.3660378

GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA

ACM ISBN 979-8-4007-0605-9/24/06
https://doi.org/10.1145/3649476.3660378

1 INTRODUCTION

Integrated circuit (IC) applications encompass a broad range, spanning from small IoT devices to large, complex multicore processors or System-on-Chips (SoCs). In practice, these are designed in hardware description languages (HDL), such as VHDL, Verilog or SystemVerilog at the register transfer level (RTL). However, while designing these ICs, especially the large and complex designs, bugs are introduced which can potentially compromise the security of these devices [5]. Every design undergoes rigorous verification to ensure security, robustness and reliability.

One of the most common verification techniques used to verify the properties of these RTL designs is *Assertion-Checking* [20]. This method utilizes specifications coded into the RTL design of hardware intellectual properties (IPs) to assert the properties and thereby verify them. The focus of each assertion is to verify individual functions and critical logic in the hardware. However, the correctness and coverage of this verification method relies on two key factors: (a) defining properties, and (b) generating correct assertions for the said properties. Defining properties for specific designs under tests (DUTs) poses a challenging task. Conventional approaches rely on designers' experience and proficiency, which are both time-intensive and prone to human errors, thus lacking robustness. The assertions developed should comprehensively cover the defined properties. Generating these assertions for each DUT is a demanding task that requires the developer's expertise.

Security assertions play a crucial role in identifying and mitigating security vulnerabilities, while functional behavior assertions validate the proper functioning of the DUT. The coverage of a functional behavioral check may be adequate for a DUT, but security verification cannot be considered satisfactory by coverage alone. Ensuring security requires knowledge of the DUT, its use case, and the potential weaknesses they possess. Prominent organizations like RISC-V, MSP, Arduino, etc., offer comprehensive documentation that details the functionalities, use cases, and threat models for their design IPs. Hence, developing a methodology for generating security assertions from DUT documentation is crucial to enhance the robustness and expedite the security verification process. Given comprehensive documentation, we posit that assertions for a DUT can be generated using the process depicted in Figure 1. By using large language models (LLMs), the documentation for the DUT can be analyzed and be processed to define properties and thereby generate assertion files that can be bound to the RTL designs. To this end, we aim to develop a natural language-based framework that can extract information from the documentation provided with

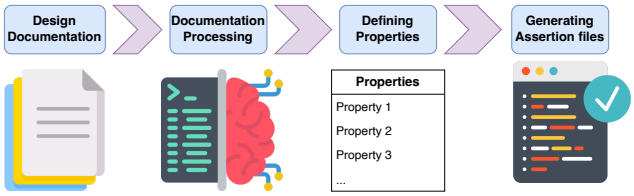

Figure 1: Assertion generation from design documentation.

the design, define security properties, and subsequently generate assertions files (*.sva files) written in *SystemVerilog*. These files can then be bound to the DUTs to apply security assertion checking.

In this paper, we propose, Assert-O, an LLM-based assertion generation framework that can generate assertions from hardware documentation and optimize them, aiming to minimize overhead in security verification processes reliant on assertion checks. To initiate the process, a dataset is constructed to fine-tune the "Falcon-40B" [2] pre-trained open-source model for generating properties. Subsequently, the model undergoes training to prioritize assertions, facilitating efficient overhead optimization. The framework consolidates properties to enhance comprehensiveness, followed by the computation of CVSS scores for severity assessment based on the IP type. Contributions of this paper can be summarized as follows:

- We propose an open-source LLM-based framework that can generate security properties, optimize them, and assign a severity level for the generated properties. This enables automating and optimizing the verification process.
- We compile a dataset to incorporate domain-specific knowledge into open-source models in order to generate properties and assertions.
- We further implement a merging methodology that merges properties based on the operation and registers names.
- When evaluated on modules from OpenTitan, Assert-O successfully extracted 183 properties, which were then optimized and reduced to 138, showcasing a reduction of 24.6%.

2 BACKGROUND

This section provides background on hardware security verification and *large language models* which are required in our proposed method, Assert-O.

2.1 Hardware Security and Verification

A plethora of techniques have been developed to ensure the security of software applications, either through the source code or binary operations [5]. However, the availability of commercial EDA tools, which are specifically crafted for hardware security, is rare. Hence, researchers have recently focused on developing tools and methodologies to ensure hardware security. Next, we will provide an overview of recent hardware security verification approaches and hardware design documentation.

Various hardware verification techniques have been proposed, including formal verification, information flow tracking, and fuzzing. These techniques either use a golden reference model (GRM) or assertions to verify the target DUV. While GRM-based verification is a widely used approach, it does not provide specific protocol checking. Furthermore, in the case of security verification, GRM-based techniques are inefficient. This is due to the construction of GRMs being focused primarily on the functional correctness of the DUV. Therefore, it is a common practice to use property definition in the hardware design process to ensure security. Furthermore, properties are also used in formal methods for both security and functional verification.

In summary, all existing security verification techniques need robust security properties to ensure the trustworthiness and robustness of the RTL. Therefore, to validate the entire SoC comprehensively, we require an effective security property generation technique that provides design assertions or operating constraints. Hence, the lack of efficient security property generation methods is the bottleneck to most hardware security verification approaches.

2.2 Large Language Models (LLM)

LLMs are cutting-edge artificial intelligence systems built on transformer neural networks and trained on massive datasets, allowing them to understand and generate human-like text with remarkable accuracy and fluency. Examples of popular LLMs include OpenAI's GPT series and Google's BERT [6, 16]. LLMs excel in tasks such as language translation, text summarization, and sentiment analysis, thanks to their contextual understanding enabled by attention mechanisms. Ongoing research aims to harness LLMs' potential to revolutionize natural language processing applications, including chatbots, virtual assistants, and content generation.

LLMs have been used for various applications in the hardware security domain, including code generation, verification, and fortification (*e.g.*, [9, 11, 17, 18]). However, off-the-shelf open-source LLMs are not adopted to the domain-specific tasks like security property generation. Therefore, these models are typically employed either by prompt engineering or by fine-tuning them for domain-specific tasks.

3 RELATED WORKS

Built on the Transformer architecture, Large Language Models (LLMs) have become significant in the domain of natural language processing [4]. These models, having undergone extensive pre-training on expansive textual datasets, demonstrate exceptional proficiency in generating and comprehending human language and thus have multiple applications [7, 17]. In one study, LLMs are employed to generate SystemVerilog assertions for hardware security verification, using natural language prompts derived from code comments [8]. Recently, LLMs have been deployed for the purpose of code generation, a process where executable code is automatically created from specifications written in natural language [15]. One study focuses on fine-tuning and assessing LLMs for their ability to generate syntactically and functionally correct Verilog code [19]. In another study, the authors have developed a fortification component that employs LLMs to automatically generate and insert additional design code that mitigates power side-channel leakage by employing Boolean Masking [18]. One of the works leverages LLMs to develop a specialized variant tailored for the hardware domain [11]. This application of LLMs showcases their capability to parse and interpret complex technical documentation, extracting specific security-related properties essential for validating the security of System-on-Chips. Another study investigates the application of LLMS, like OpenAI's ChatGPT and Google's Bard, in the

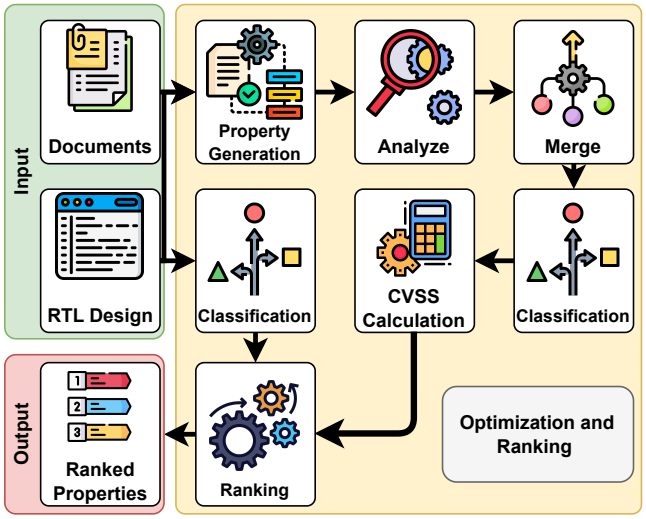

Figure 2: Workflow of Assert-O

context of hardware design, focusing on their potential to automate the translation of natural language specifications into HDL such as Verilog [3]. In an alternate study, LLMs are used for zero-shot vulnerability repair, where they are tasked with generating repaired versions of insecure code, addressing challenges in prompt design and assessing their performance across real-world security bug scenarios [14]. Another research utilizes LLMs to automatically repair security-relevant bugs in hardware designs through a framework that facilitates prompt engineering and parameter optimization, demonstrating that LLMs can effectively outperform existing hardware bug repair tools [1]. Authors employ LLMs to aid in reverse engineering tasks by interpreting and explaining code through a combination of open-ended questioning and a true/false quiz framework, revealing significant potential despite current limitations in zero-shot reverse engineering in [13].

4 ASSERT-O ARCHITECTURE

In this section, we propose Assert-O, an NLP-based property generation and optimization framework that generates security properties from documentation and optimizes the number of properties by merging and ranking the generated properties.

In order to generate security properties from documentation, Assert-O follows the steps depicted in Fig. 2. First, Assert-O takes the documentation and the RTL design as input. The RTL design is classified into several categories based on their role in the overall system. From the documentation and the RTL design, Assert-O generates security properties. When extracting security properties from the design, the properties are determined by the following two factors: (1) they must have a condition paired with the requisite assignments or operations, and (2) violation of the said property must incur a security violation (*e.g.*, information leakage, side-channel attack, secure-data corruption, privilege escalation, etc.). The generated properties are then analyzed and merged based on the operation and the type of registers involved. Next, the merged properties are sorted into various categories to determine the CVSS score for each. Using the CVSS score and the category of the RTL design, the properties are ranked and categorized into five categories based on the severity of the impact if they were to be breached.

In the subsequent sections, we will elaborate on the steps involved in generating the optimized properties through Assert-O followed by their optimization process.

4.1 Property Generation

To generate security properties from documentation, it is essential to use a decoder-only pre-trained model that undergoes fine-tuning to specialize in security property generation. For this purpose, we utilize the pre-trained model, "*Falcon-40B*" as the base model [2].

To enhance Assert-O's ability to recognize and generate security properties from documentation, the initial step involves generating security properties from a given SoC's documentation. To fine-tune Assert-O for this purpose, a dataset comprising approximately 70 documentation segments sourced from various hardware IPs within the OpenTitan SoC has been compiled. The entries include documentation segments explaining operations to prevent security vulnerabilities of corresponding hardware IPs. The security properties encompass the following traits that relate to security vulnerabilities resulting from the violation of these properties: (a) information leakage, (b) data corruption, (c) privilege escalation, (d) bypassing security protocols, and (e) susceptibility to side-channel and fault attacks. For example, the AES module employs a first-order masking policy to make the AES operation resilient against power side-channel attacks. The complete masking operation is detailed in the documentation. This qualifies as a security property, as a violation of the masking operation may lead to both information leakage and the module's susceptibility to side-channel attacks. With the primary set of documentation segments collected, the dataset is subsequently augmented with parts of these collected segments and their corresponding security properties. Furthermore, segments lacking security properties are included in the dataset to diversify the model and prevent it from generating security properties from all inputs. The original segments are also rephrased to introduce variety to the fine-tuning dataset.

Upon the generation of the dataset, the model is deployed to extract properties from the documentation. These properties often include overlapping checks on operation execution. For instance, when verifying masking operations in AES, properties may cover flag control, masking activation, and the operation itself. However, flag control and masking activation share similar checkpoints, allowing for consolidation into a single comprehensive property rather than two separate ones.

4.2 Design Classification

Security properties of IPs in an SoC can be prioritized based on their role in the system. Therefore, it is important to identify the role of each IP in the system, such as memory blocks, BUS, crypto cores, processors, random number generators, etc., prior to ranking the security properties for the design.

To determine the role of IPs within the system, Assert-O initially constructs the design hierarchy, thereby establishing the location and interconnections of the IPs. With this hierarchy in place, Assert-O identifies the BUS of the system. Next, Assert-O uses the

Algorithm 1: Optimization

Input: Generated Property list, P_{list}
Output: Ranked property list, P_{ranked}

```
1  for P in P_list do
2  |  Classify_operation of P;
3  |  Classify_register of P;
4  end
5  forall Operation do
6  |  P_M ← Merge(P ∈ P_list);
7  end
8  forall Register Types do
9  |  P_M ← Merge(P ∈ P_M);
10 end
11 forall P in P_M do
12 |  forall Classes in CVSS do
13 |  |  Classify_class of P;
14 |  end
15 |  Calculate_CVSS(P);
16 |  Calculate_Final(P, IP, IP_role);
17 end
```

documentation to identify each IP's function in the system. The subsequent task involves identifying the specific modules comprising each IP, accomplished through consideration of module names and their parent-child relationships. Next, Assert-O groups and assigns the modules a role such as memory block, crypto core, peripheral, debug interface, or accelerator. This facilitates the assignment of weights to properties based on their vulnerability associations.

4.3 Optimization

To optimize generated properties, Assert-O follows Algorithm 1. Assert-O takes the generated properties, P_{list} as input. Next, Assert-O classifies all the properties, $P \in P_{list}$ based on their operation and the type of registers (*i.e.*, control, flag, data, etc.) (line 1 to line 4). After the properties are all classified, Assert-O merges all the properties that perform a similar type of operation and constructs the list P_M (line 5 to line 7). Subsequently, Assert-O merges all the properties, $P \in P_M$ that are using similar types of registers and reconstructs P_M (line 8 to line 10). Upon merging the properties, Assert-O proceeds to rank the merged properties. To rank the properties, Assert-O first classifies each property, $P \in P_M$ to the classes used in calculating CVSS scores(line 12 to line 14). Upon classification of the properties, Assert-O calculates CVSS scores of the properties (line 15). Next, Assert-O calculates the final scores that represent the severity of vulnerability associated with their corresponding properties (line 16). Finally, Assert-O sorts the properties based on the final scores assigned to them (line 18).

The following sections provide details on how Assert-O is trained to optimize the generated properties.

```
1  logic err_q;
2  always_ff @(posedge clk_i or negedge rst_ni) begin
3    if (!rst_ni) begin
4      err_q <= '0;
5    end else if (intg_err || reg_we_err) begin
6      err_q <= 1'b1;
7    end
8  end
9  // integrity error output is permanent and should
     be used for alert generation
10 // register errors are transactional
11 assign intg_err_o = err_q | intg_err | reg_we_err;
```

Listing 1: Integrity error function for PWM.

4.3.1 Merging. Upon generating security properties from documentation, Assert-O merges the properties to reduce the number of properties. To train Assert-O to merge properties, another dataset is created to fine-tune the base model and make another fine-tuned model. This model merges the security properties by using two metrics: (a) the type of operation the properties are describing and (b) the type of registers involved in the operations.

Security properties detail operations run by a module to bolster the security of that module. Moreover, each module in the design of an SoC has a number of registers that can be classified into several types, *e.g.*, data registers, control registers, flag registers, status registers, etc. To merge properties based on the operation and the type of registers involved, we take the properties used for training the properties generator, merge them, and create another dataset. For example, in the AES design, a finite state machine (FSM) is implemented using sparse encoding. The muxes, handshakes, etc., are also controlled by sparse encodings. These properties elaborate on the use of the encoding method on *control* signals. Therefore, these properties can be merged to make one single property for the AES module regarding the encoding procedure.

To illustrate the merging process, we use the toy example illustrated in Listing 1. Here, a code snippet for an asynchronous reset operation is implemented. For this code-snippet, the following properties displayed in Listing 2 are generated:

```
1  property assert_rst_0;
2      !rst_ni |-> err_q == 0;    // the value of \
       textit{err\_q} should be set to 0.
3  endproperty
4
5  property assert_rst_0;
6      !rst\_ni |-> $stable(intg_err);    // the value
       of intg_err should remain the same.
7  endproperty
8
9  property assert_rst_0;
10     !rst\_ni |-> $stable(reg_we_err);    // value of
       reg_we_err should not change.
11 endproperty
```

Listing 2: Generated Properties.

These three properties, each operating on distinct registers, can be combined into a single comprehensive property, as depicted in Listing 3. This merging is achieved by matching their operations, *i.e.*, by aligning the condition and assignment checks. Since all three properties share the same condition, they can be consolidated by ANDing their checks into one complete property.

```
1  property assert_rst_0;
2      !rst_ni |-> err_q == 0  && $stable(intg_err) &&
         $stable(reg_we_err);
3  endproperty
```

Listing 3: Merged Property.

Merging reduces the number of properties that can subsequently be ranked based on their importance and the severity of the security risk they pose to the overall system.

4.3.2 Ranking. In order to optimize the assertion implementation overhead in the code, a method for ranking the properties based on their importance to the system is necessary. Therefore, a comprehensive ranking system was conceptualized to evaluate and prioritize the merged properties based on their respective Common Vulnerability Scoring System (CVSS) scores and the type of security vulnerability associated with that property. The CVSS score is a numerical representation of a vulnerability's severity that can be used to rank the security properties [10]. However, the same type of vulnerability can carry different weights to different modules of a design. For example, information leakage is more severe for cryptographic cores than for the peripherals of the system. On the other hand, privilege control is a severe threat to the peripheral IPs. Therefore, it is important to To rank the properties, the process is divided into three different subprocesses: (a) CVSS score calculation, (b) assigning weight to the properties based on the module they belong to, and (c) finally assigning the overall score representing the severity for ranking the properties.

CVSS Score Calculation. The Common Vulnerability Scoring System (CVSS) score is a structured assessment framework that measures the severity of security vulnerabilities present within software or systems [10]. It achieves this through the classification of vulnerabilities based on their characteristics. The CVSS score consists of three primary groups, namely the Base, Temporal, and Environmental metrics, each encompassing a set of attributes. The Base group defines intrinsic vulnerability characteristics, the Temporal group considers time-related factors, and the Environmental group contextualizes vulnerabilities to specific user environments. These groups are further divided into: Attack Vector, Complexity, Requirements, Privileges, User Interaction, Scope, Confidentiality, Integrity, and Availability.

To determine the CVSS score of a property, Assert-O has been trained to categorize each property into specific classes. For instance, a property can be linked to an attack vector type, such as '*Network*,' '*Adjacent*,' '*Local*,' or '*Physical*.' Specifically, properties that describe the concealment of first-order differential power side-channel attacks are classified as being associated with a '*Physical*' attack vector.

In order to enhance the classifier's ability to accurately categorize properties into their corresponding attack vectors, attack complexity levels, attack requirements, privilege requirements, user interactions, and security class levels for confidentiality, integrity, and availability, we assign a CVSS vector to the combined properties within the dataset. This CVSS vector encapsulates the mentioned property classes. Subsequently, by utilizing the refined classifier, we can determine the classes for each property and generate a CVSS vector, which enables us to compute the associated CVSS score for that specific property.

Table 1 illustrates a sample row from the training dataset, showcasing a property of the OpenTitan Big Number (OTBN) accelerator. In this instance, the 'Attack Vector (AV)' is denoted as 1, indicating that exploiting this property necessitates local execution of the attack. Likewise, the 'Privileges Required' is assigned a value of 2, signifying a high level of user privilege needed for a successful attack. In summary, the CVSS vector for exploiting this property is as follows:

CVSS:4.0/AV:L/AC:H/AT:P/PR:H/UI:A/VC:H/VI:L/VA:L/SC:H/SI:L/SA:L

Using this vector, the overall score for the given property is 5.9 using the CVSS v4.0 standard.

Assigning Weight to Property for a Module. In order to rank properties based on the module's role in the overarching system, the ranking system of Assert-O needs to be trained to assign weights to the properties. In this step, Assert-O is trained with a dataset that classifies each property with the types of security vulnerability it is associated with.

Assert-O is trained with a dataset that lists the module's role in the overall system, the CVSS score of the property, and the type of vulnerability. The type of vulnerability can fall into the following categories: (1) information leakage, (2) information corruption, (3) privilege control, (4) metastability of states, (5) protocol control, and (6) error handling. Assert-O is trained with this information to assign the properties a category (*i.e.*, critical, high, medium, low) to rank the properties.

In Table 1, the IP OTBN serves as an accelerator for cryptographic cores handling operations such as RSA or Elliptic Curve Cryptography (ECC). This specific property concerns the clearance of data residue post-operation, falling under the category of information leakage violations. Given the critical nature of information leakage from cryptographic cores, this property is assigned the highest weight value of 1.

5 EVALUATION AND RESULTS

5.1 Experimental Setup

In order to evaluate the efficacy and versatility of Assert-O in generating security properties, the benchmarks are required to have comprehensive documentation, as Assert-O necessitates documentation for security vulnerabilities for the given design. Furthermore, the code base must offer functionalities such as Finite State Machines (FSM) and resets in synchronous and asynchronous operation modes to enhance the model's applicability across various designs. diverse implementation of these functionalities in the code base ensures Assert-O's applicability on different designs. To this end, we evaluate Assert-O using the OpenTitan SoC [12] since it: (1) is an open-source industry-grade design, (2) has well-detailed

Table 1: Example Dataset Row for Property and IP role Classification.

IP Name	Property	Attack Vector	Attack Complexity	Attack Requirements	Privileges Required	User Interaction	Scope	Confidentiality	Integrity	Availability	IP role
OTBN	Secure wiping, issued by 'SEC_WIPE_DMEM', is performed by securely replacing the memory scrambling key, making all data stored in the memory unusable.	1	1	1	2	2	2	2	1	1	Crypto

Table 2: Property Generation and Merging Summary

IP Name	Sentences in Document	Properties Generated	Properties after Merging	Severity of Vulnerabilities			
				Critical	High	Medium	Low
PWM	129	36	26	-	4	14	8
ROM Control	69	22	17	5	3	6	3
UART	83	27	19	7	4	5	3
AES	85	41	32	7	8	12	5
Reset Manager	73	45	37	9	11	17	-
Pattern Generator	22	12	7	-	2	3	2
Total	461	183	138	28	32	57	21

documentation, and (3) includes a wide array of implementation of FSMs and resets with both synchronous and asynchronous operation mode. The following IPs were utilized from OpenTitan as benchmarks in our experiments: *PWM, ROM Control, UART, AES, Reset Manager*, and *Pattern generator*. Our experimental platform is composed of an Nvidia DGX server equipped with four Nvidia A-100 GPU cores and 1TB memory.

5.2 Assert-O Evaluation

Assert-O was evaluated by analyzing the documentation of six distinct OpenTitan IPs that had not been employed in the fine-tuning of Assert-O. The IPs utilized for this evaluation encompassed a Pulse Width Modulator, ROM control, UART, AES, Reset Manager, and Pattern Generator. As illustrated in Table 2, the combined documentation for these IPs comprises a total of 461 sentences, with individual documents ranging in length from 22 to 129 sentences. From the documentation, Assert-O initially extracted 183 properties, but following a merging operation, the total properties were reduced to 138. Among these 138 properties, 28 are classified as having a critical vulnerability potential, while 32 are categorized as having a highly severe vulnerability potential.

From Table 2, it can be observed that the IP, PWM has the most significant merging operation reducing 36 generated properties to 26 properties. This is due to the functions of the PWM. As the protocols of the PWM are mostly similar and have overlapping functionalities, the generated properties for each security operation assert similar operation checks and use the same registers to verify the properties. Merging seems to reduce the security properties of UART and AES despite their different security protocols. This integration arises from how these properties are coded. In the OpenTitan codebase, common flags for security checks are utilized, enabling the assertion of these properties based on flag values. This allows merging operations to be more effective for these IPs. When the categories based on the severity of these IPs are observed, it can be seen that the pattern generator and PWM modules do not have any critical-level security vulnerability. This is due to the IPs being peripheral controllers. These IPs do not have any functionality that the adversary can exploit to execute any critical error.

On the other hand, none of the security properties pertaining to the reset manager fall into low-severity vulnerability. The primary reason behind this is that the reset manager controls the reset activities of the SoC. Therefore, all the security properties have a high impact on the overall system.

Overall, Assert-O successfully extracted a comprehensive set of 183 properties from an initial dataset comprising 461 lines of text. Employing a merging operation, Assert-O efficiently condensed these properties into a subset of 138, representing a notable reduction of 24.6%. Subsequently, these merged properties are categorized into four distinct groups. The distribution of the severity impact of these properties illustrates security implications associated with the corresponding IPs under scrutiny. Notably, among these categories, the 'Reset Manager' emerged as the element with the most significant security impact on the system, closely followed by the UART and AES components. These categories optimize verification, allowing for the exclusion of low-severity vulnerabilities to minimize penalties.

6 CONCLUSION

In summary, our study highlights the importance of hardware security validation within modern SoC designs, which face vulnerabilities stemming from integrated IP cores. Traditional validation methods, relying solely on manual expertise, are not scalable and time-intensive as they are prone to human errors. To this end, in this paper we propose, Assert-O, a specialized automated framework utilizing LLM, we aim to extract and enhance security properties from SoC documentation, reducing the need for manual intervention by prioritizing critical security properties. Through evaluation on OpenTitan IPs, Assert-O effectively reduced a comprehensive set of 183 properties to 138, marking a 24.6% decrease. Further categorization revealed the Reset Manager IP as the most impactful, followed closely by UART and AES components. Overall, Assert-O offers a promising approach to bolster hardware security validation in SoC designs, paving the way for more robust and secure computing systems.

ACKNOWLEDGMENTS

This research is supported by the Technology Innovation Institute (TII), Abu Dhabi, UAE.

REFERENCES

[1] Baleegh Ahmad et al. 2023. Fixing Hardware Security Bugs with Large Language Models. arXiv:2302.01215 [cs.CR]
[2] Ebtesam Almazrouei et al. 2023. The Falcon Series of Language Models: Towards Open Frontier Models. (2023).
[3] Jason Blocklove et al. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. *arXiv preprint arXiv:2305.13243* (2023).
[4] Yupeng Chang et al. 2023. A survey on evaluation of large language models. *arXiv preprint arXiv:2307.03109* (2023).

[5] Ghada Dessouky et al. 2019. {HardFails}: Insights into {Software-Exploitable} Hardware Bugs. In *28th USENIX Security Symposium (USENIX Security 19)*. 213–230.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[7] Jean Kaddour et al. 2023. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169* (2023).

[8] Rahul Kande et al. 2023. LLM-assisted Generation of Hardware Assertions. arXiv:2306.14027 [cs.CR]

[9] Mingjie Liu et al. 2023. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176* (2023).

[10] Peter Mell et al. 2007. A complete guide to the common vulnerability scoring system version 2.0. In *Published by FIRST-forum of incident response and security teams*, Vol. 1. 23.

[11] Xingyu Meng et al. 2023. Unlocking Hardware Security Assurance: The Potential of LLMs. arXiv:2308.11042 [cs.CR]

[12] OpenTitan [n. d.]. Documentation | OpenTitan. https://opentitan.org/documentation/index.html

[13] Hammond Pearce et al. 2022. Pop Quiz! Can a Large Language Model Help With Reverse Engineering? arXiv:2202.01142 [cs.SE]

[14] Hammond Pearce et al. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2339–2356. https://doi.org/10.1109/SP46215.2023.10179324

[15] Gabriel Poesia et al. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227* (2022).

[16] Konstantinos I Roumeliotis and Nikolaos D Tselikas. 2023. Chatgpt and open-ai models: A preliminary review. *Future Internet* 15, 6 (2023), 192.

[17] Dipayan Saha et al. 2023. Llm for soc security: A paradigm shift. *arXiv preprint arXiv:2310.06046* (2023).

[18] Amisha Srivastava et al. 2023. SCAR: Power Side-Channel Analysis at RTL-Level. arXiv:2310.06257 [cs.CR]

[19] Shailja Thakur et al. 2023. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. https://doi.org/10.23919/DATE56975.2023.10137086

[20] Hasini Witharana et al. 2022. A survey on assertion-based hardware verification. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–33.