Security Verification in Practice: Lessons from Pre-Silicon Analysis of SoC Subsystems

Yashwanth Kumar A R, Rachana Maitra
Marvell Technology, Inc
yarajeshkuma@marvell.com, rmaitra@marvell.com

***Abstract*- As the complexity of System-on-Chip (SoC) architectures increases, so does the urgency to address security vulnerabilities early in the design cycle. Traditional functional verification alone is insufficient to detect subtle security flaws that adversaries may exploit through physical, firmware, or protocol-level attacks. This paper presents a practical framework for pre-silicon security verification that integrates static and dynamic techniques to uncover vulnerabilities in the pre-silicon stage, specifically, before synthesis. The methodology combines security-focused linting to detect security-sensitive weaknesses in RTL code, dynamic information flow tracking to ensure asset confidentiality and integrity, and side-channel analysis to identify potential leakage paths through power analysis. The evaluation of these techniques was conducted across various subsystems, selected for their applicability to practical threat models and diverse architectures. The paper also discusses the challenges encountered, including limitations in structural support, abstraction mismatches, and the need for enhanced collaboration between design and verification teams. A key observation is that security verification must be asset-centric and layered, with verification goals aligned to confidentiality, integrity, and availability (CIA). The framework not only helps shift security left in the development cycle but also prepares teams to think proactively about threat modeling and test coverage. Future work will focus on expanding the methodology to additional subsystems, enhancing automation, and aligning security vulnerability checks with industry standards, such as the Common Weakness Enumeration (CWE).**

Keywords: SoC (System on Chip), pre-silicon stage, security verification, shift-left approach, CIA, CWE.

I. Introduction

The increasing complexity of modern System-on-Chip (SoC) and Register Transfer Level (RTL) designs has introduced a wide array of security challenges that traditional verification methodologies are unable to identify and solve. As SoCs integrate more heterogeneous components, the potential for security vulnerabilities—ranging from data leakage to privilege escalation, grows significantly. These vulnerabilities can range from a simple omission of a “default” case statement in a Finite State Machine (FSM) to a more complex scenario where an asset remains unprotected, which can be addressed by implementing a secure boundary or an access control mechanism. If these are overlooked, they can be exploited by adversaries, leading to reputational damage, financial loss, and compromised user trust, especially in this age of artificial intelligence.

Over the course of years, security verification, particularly hardware security, has been seen as a ‘good to have’ and was an add-on. More importantly, security verification when done as early as possible will result in less damage to the security threat of a design or even a chip in the chip development flow. Security bugs discovered late in the development cycle are not only harder to fix but also more expensive to mitigate. The industry is now recognizing the need to "shift left," which can be accomplished by bringing security analysis into earlier stages of the design flow, particularly during pre-silicon verification. This shift in the mindset and action enables the identification of architectural flaws, insecure design patterns, and unintended data flows before they become deeply ingrained in the system.

Despite this increasing awareness, integrating security into traditional design verification (DV) processes remains challenging. Functional DV mainly focuses on correctness, performance, and coverage, for example, performing normal read/write tests on a block going by block specification, while security verification requires a different approach—one that is asset-focused, like testing if the asset flows to an unintended location or at an unexpected time. Security objectives, such as Confidentiality, Integrity, and Availability (CIA), must be translated into specific verification goals, which involve creating a threat model, establishing rules, and developing testing strategies centered around the asset.

Unfortunately, there are various difficulties due to the changing nature of hardware threats, the absence of standard practices, and metrics. To identify potential threat vectors, describe essential assets, and comprehend the system's security intent, verification teams frequently need to collaborate closely with designers. This cooperation is necessary to guarantee that security verification is a significant and efficient procedure. It also necessitates a change in mindset, where design, verification, and architectural teams should view security as a shared duty.

This paper focuses on a practical framework for pre-silicon security verification that complements traditional DV flows. By combining static and dynamic techniques, the framework enables early detection of vulnerabilities related to control logic, data paths, and asset exposure. It emphasizes the importance of the shift-left approach and provides insights into how security verification can be seamlessly integrated into existing workflows without disrupting productivity. The goal is to make security verification scalable and impactful, thereby ultimately contributing to more resilient designs.

II. Security Verification Methodologies

Security verification in pre-silicon design requires an asset-aware approach that goes beyond traditional functional correctness, which is based on the design specification. The methodology adopted in this work is multi-pronged, combining static and dynamic techniques to comprehensively evaluate the design's resilience against confidentiality, integrity, and availability (CIA) threats. Each technique plays a distinct role in identifying different classes of vulnerabilities, and together they form a framework for early-stage security assurance.

A. Static Security Verification

Static security-based verification primarily involves analysis of the design under question at design time, where no simulation or run time is required. This technique performs the analysis of the RTL code directly for security issues, which could use formal methods, structural checks, or code analysis. This static security approach is an add-on to the existing linting tools, which are widely used across the design teams, where they run many linting checks to detect code errors on the RTL. In this way, security-based linting would be much easier for the design and even the design verification teams to adopt.

Static security analysis must be set as the first line of defense owing to many of its advantages, as listed in the above paragraph, where they are easy to integrate into existing linting workflow, and they will not require any simulation time, unlike the techniques that are used in the dynamic-based security verification. While it has its advantages, some of the limitations of the static-based security checks are that they cannot be used to detect dynamic behavior and are limited to structural and semantic checks from a security point of view.

This involves detecting unsecured data/control paths, unreachable or undefined states, and unsafe finite state machine (FSM) transitions, among other things. When it comes to identifying design patterns that could result in by passable logic, unauthorized access, or privilege escalation, static checks are especially useful. Even teams without extensive security knowledge can utilize these tests because they are scalable, allowing for streamlined usage when integrating into current linting procedures.

B. Dynamic Security Verification

As far as the dynamic-based security verification methodology is concerned, it has several types of security vulnerability identification techniques, but this paper will talk primarily about Information Flow Tracking (IFT) as shown in [1] and side channel analysis. The information flow tracking is a particularly useful method that is much more focused on monitoring assets present in the design or IP. It makes sure that sensitive information (assets) does not leak to unsecured or unauthorized locations by simulating how it moves through the design. IFT is especially helpful for confirming that read/write permissions are correctly enforced, that data isolation is maintained across privilege domains.

This information flow analysis is primarily based on the confidentiality, integrity, and availability of the asset in question. For example, a memory has keys inside it, which are our assets. In this design, as shown in Figure 1, a secret cryptographic key stored in memory is accessible only by a crypto engine identified through a unique hardware ID (HWID) matching. Using information flow tracking analysis, we can verify the confidentiality of the key by ensuring

it never leaks to unauthorized outputs, its integrity by confirming only the authorized engine can modify it, and its availability by checking that the crypto engine can access the key when required. This demonstrates the enforcement of the CIA triad under realistic, stimulus-driven conditions.

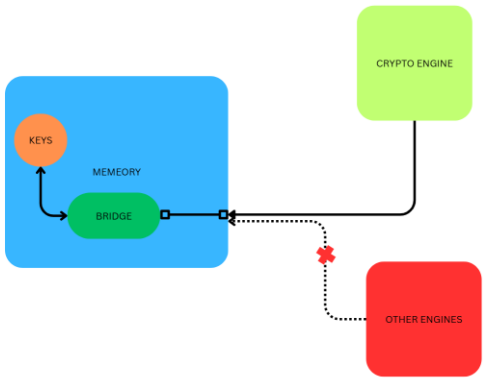

Fig. 1. Secure asset block diagram

Side-channel analysis (SCA) is another critical dynamic verification technique used to evaluate primarily the confidentiality of sensitive assets by observing indirect leakage through power consumption. SCA, in contrast to the other structural or functional checks, focuses on how data-dependent changes in switching activity may inadvertently disclose details about secret values, like cryptographic keys. SCA can be conducted in pre-silicon environments utilizing simulation-based power estimates, which analyze switching behavior under controlled stimulation to find correlations between sensitive data and internal signals.

Using the SCA approach, test vectors are injected, and the design's switching behavior is observed over time. Following that, statistical methods are used to see whether any internal signals show patterns that depend on data and could be used by an attacker. This method works especially well for finding partial key leakage or accidental connections between secure and non-secure domains that might not be visible through conventional testing. SCA makes it possible to identify potential leakage channels early on by simulating realistic attack scenarios. This lets designers implement countermeasures like masking, random noise generation, or logic randomization prior to tape-out. Although SCA is much more beneficial when it is done in the post-silicon phase, it will be beneficial to implement this in the pre-silicon phase as well, since it allows early detection of potential leaking signals and paths, which reduces costly silicon re-spins.

III. Evaluation Framework

A wide range of subsystems was chosen for evaluation to validate the proposed security verification technique. The idea of maximizing coverage across various security-relevant logic types, such as cryptographic engines, control-oriented finite state machines (FSMs), and data management units like NVMe queue manager blocks, served as the basis for the selection. These blocks were chosen based on their criticality to the system's security posture and their role in enforcing access control or managing sensitive data. The objective is to expand the security robustness to the extended components and assets, in addition to strengthening security for the Root of Trust (RoT) components.

The ideal recommendation is to deploy both static as well as dynamic methods to at least a security-sensitive block, if not for most of the SoC design. This is because security is as strong as the weakest link present in the system. Although we acknowledge that it takes huge time and effort to accomplish a thorough security verification across most of the SoC blocks, there should, however, be no compromise in the analysis of asset-laden blocks, critical processors, etc., using both static and dynamic verification methods.

Another rule of thumb is to use the proposed security linting throughout the course of RTL development, as the designers would already be using the regular linting in parallel during RTL development. As far as the dynamic methods are concerned, especially IFT analysis, since it requires simulation, it is best to deploy these techniques when

the functional tests are at least 70% completed or before the RTL freeze. This is because IFT relies on functional tests to activate relevant signals and exercise the block meaningfully, so that the tagging of concerned signals would be meaningful.

Defining the verification scope was also a vital stage in the process. The methodology focused on scoping the analysis to relevant sub-hierarchies, rather than analyzing the entire SoC. A generic example is the path from the external agent through the access control logic to asset-containing modules. This hierarchical scoping ensured that the analysis remained tractable while still capturing meaningful security properties.

This structured evaluation framework enabled the application of the security verification methodology in a scalable and regressionable manner. It also highlighted the importance of early collaboration with design and architecture teams to identify critical assets, define threat models, and align verification goals with security requirements.

IV. Case Studies

Readers should note that the authors evaluated their methods on proprietary designs with both no known bugs and artificial bugs to confirm basic functionality and robustness. The aim was to show that the techniques can detect known issues and reliably handle controlled tests, proving their practical value.

A. Security Linting Evaluation

Static security linting was applied to a control-oriented subsystem with multiple FSMs and data/control path logic. The focus was on identifying vulnerabilities that could arise from unsafe state transitions, unreachable states, ambiguous reset behavior, and other subtle security violations. Additionally, the linting process flagged several FSMs with undefined default states and 'don't care' values, such as X, which could lead to unpredictable behavior under corner-case conditions.

While general linting tools might miss subtle failures that are security sensitive, the latest ones could catch some of the issues mentioned above. However, linting from a security point of view would provide much more insight into how the reported violation could lead to a security vulnerability. Static security linting has revealed a class of vulnerabilities that include incorrect usage of bytewise or bitwise control logic in configuration registers. This presents two significant security concerns. It first makes data-dependent behavior possible, including time or power-based attacks, which can be exploited by side-channel analysis. Second, such logic may allow unintended reconfiguration immediately after reset, before enforcement of access controls—posing potential security risks under specific conditions. If not properly mitigated, this behavior could introduce vulnerabilities that affect system integrity or access control enforcement. This is highlighted in Figure 2, where the left snippet is the original RTL, and the one on the right is a way to mitigate the vulnerability.

```
// Original RTL

for (int i = 0; i < 8; i++) begin
  if (wr_en && config_select && wr_data[i])
begin
    ctrl_reg[i] <= wr_data[i];
  end
end
```

```
// Fix for the issue

reg config_lock;  // Lock to block config writes until secure state

// Lock logic: starts locked (1), unlocks after secure init
always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n)
    config_lock <= 1'b1;              // Locked after reset
  else if (secure_init_done)
    config_lock <= 1'b0;              // Unlock after secure init
end

// Protected config write: only if unlocked (!config_lock)
for (int i = 0; i < 8; i++) begin
  if (wr_en && config_select && !config_lock && wr_data[i]) begin
    ctrl_reg[i] <= wr_data[i];
  end
end
```

Fig. 2. RTL code issue and fix

Although commonly used for write-one-to-set or write-one-to-clear register types, the original logic lacks temporal access control and assumes that all updates are trusted. Static security analysis highlights such issues that a regular linting tool will miss by identifying post-reset configurability of critical signals and flagging granular, bit-level comparisons as side-channel-prone structures. Now, this is up to the designer or verification engineer to waive or debug this flagging, understanding the entire design context. Issues like these could also be caught during the

simulation verification phase, but they could be coupled with other issues, and the DV team can ignore these as mere warnings.

This highlights the value of security-focused linting in identifying subtle vulnerabilities that may not be evident through functional testing alone. In some cases, the analysis framework was able to associate detected issues with standardized vulnerability classes, such as those defined in the Common Weakness Enumeration (CWE) by MITRE [2]. This capability enhances clarity regarding the nature of the security concern and supports a broader understanding, even among engineers new to hardware security. These results reinforce the value of static analysis as a first-line defense, especially when applied early in the RTL development cycle.

B. IFT evaluation

Information Flow Tracking (IFT) was applied to a subsystem containing multiple assets guarded by a control logic in the entry of that subsystem. These assets can be configured only by a secure processor that has to satisfy some strict policies of the control logic. The goal was to verify that any non-secure processor should not influence the assets, and that sensitive data did not leak to unauthorized outputs. The analysis was scoped hierarchically to trace signal flow from external agents through control logic into asset-containing modules.

As noted in the disclaimer and due to proprietary restrictions, specific design element names and actual bugs cannot be disclosed. Nevertheless, a real-time scenario provoked as a litmus test effectively offered a benchmark for evaluating our analytical methodology. Within this case, a security strategy was evaluated using IFT to trace AXI signals passing through a control logic block en route to a protected subsystem containing security-sensitive assets, as shown in Figure 3. Initially, the analysis flagged this as a potential security concern, as access to the assets appeared permitted. However, further investigation revealed that the control logic enforces strict access control policies, and any AXI signal emerging from it is, by design, authorized for read/write operations to the subsystem, and the check should have been done initially on any AXI coming into the control logic.

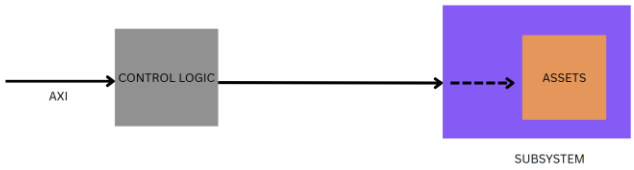

Fig. 3. Subsystem with a control logic

In a separate evaluation conducted on another subsystem, the objective was to validate that no data leakage occurred along the data path during access to the eHSM. While the analysis confirmed the robustness of the design and did not reveal any functional issues, it led to a valuable architectural insight that prompted the addition of further protective measures. This recommendation stemmed from an asset-centric security perspective and may not have surfaced without the application of dynamic verification techniques. The outcome underscores the role of such methodologies in enhancing design resilience and uncovering opportunities for proactive security hardening.

These scenarios necessitate the importance of contextual understanding of design intent in rule development and validation. Furthermore, the capability of IFT to identify these nuanced actions in a litmus test stands as evidence of the importance of dynamic security evaluation in revealing real bugs as well. It illustrates that such techniques can reveal immediate security concerns that could remain unnoticed by standard functional testing.

C. Side Channel Analysis

Side-channel analysis was performed on a subsystem implementing a cryptographic operation. The evaluation was focused on power-based leakage detection using simulation-driven switching activity analysis. The methodology involved injecting a stimulus and monitoring internal signal transitions to identify a correlation with sensitive data. Figure 4 illustrates a side-channel evaluation process that links the design behavior to physical leakage observations, as adapted conceptually from [3]

This workflow involved trace generation, where initially, simulation traces were generated from RTL-level testbenches of the targeted design. The framework extracted signal activity focusing on critical modules such as SubBytes, ShiftRows, and MixColumns. Input parameters (plaintext, key) and output parameters (ciphertext) were defined, and triggers were configured to capture transitions during encryption operations. This was a stimulus-based simulation. Further, during leakage analysis, it employed multiple statistical distinguishers, including Correlation Power Analysis (CPA), Normalized Inter-Class Variance (NICV). The analysis was performed on both unprotected and masked AES implementations.

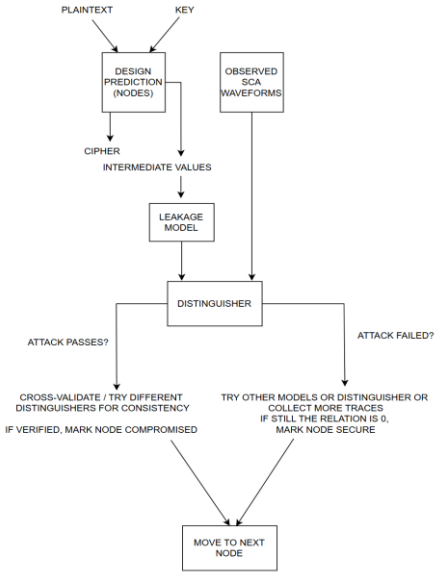

Fig. 4. SCA workflow

The plaintext and key are inputs in this case to the AES, and intermediate node values are extracted to form predictions based on a chosen leakage model (e.g., Hamming weight or distance). These predictions are compared with measured side-channel waveforms using any distinguisher, such as DPA or CPA, to determine the correlation between predicted and observed leakage values. If the attack passes, which means there is a significant correlation, cross-validation with different distinguishers confirms whether the node is compromised. If no relation is found, alternative models, distinguishers, or additional traces are assessed, and if there is persistent failure, then the node is secure. This iterative node-by-node analysis continues across the design to localize leakage points.

V. Results

The proposed security verification methodology was used across three different subsystems, each representing a variety of security-critical logics such as control-oriented blocks, cryptographic engines, etc. The evaluation helped validate the effectiveness of static and dynamic techniques.

A. Security Linting Evaluation

The static security analysis showed multiple issues, especially with the control logic of FSMs, and detected missing default case statements, unsafe state transitions, and don't care values. More importantly, it flagged certain critical issues such as bit-by-bit/byte-by-byte comparison, post-reset control logic configuration, leaving IP instantiation ports unconnected, etc., which a normal linting tool would not have captured, especially the subtle security issues.

B. IFT evaluation

Although the CIA was mapped in case of a violation using security linting, IFT was used to verify the CIA in an asset-centric approach, such as the case of assets within subsystems. The methodology highlighted a leaking access path resulted due to incorrect scoping, which permitted external writes to the asset. This technique also led to architectural improvements, such as the recommendation to implement a lock mechanism for sensitive logic—an insight that emerged through asset-centric dynamic evaluation.

C. Side Channel Analysis

TABLE I
Side Channel Analysis results

TEST CASE	NUMBER OF TRACES	WORKING DISTINGUISHER (DPA / CPA)	SUBKEYS RECOVERED (out of 16)	LEAKAGE DETECTED
Unprotected AES	500	NONE	0	NO
	1000	CPA	7	PARTIAL
	5000	BOTH	16	YES
	10000	BOTH	16	YES
Masked AES	500	NONE	0	NO
	1000	NONE	0	NO
	10000	NONE	0	NO

For the SCA evaluation, as shown in Table 1, it was performed on both masked and unmasked single AES design. As shown in the table, the number of subkeys that were determined using side channel analysis was related to the number of channel traces of the design. Leakage was observed at the first round of AES, particularly in the SubBytes and key mixing stages. While this was the case with an unmasked design, the SCA was unable to find any subkeys for a single masked AES design. The analysis was also dependent on using the type of distinguisher between CPA and DPA.

VI. Challenges And Observations

While this security framework has been quite useful in detecting bugs that are detrimental to the security context, there have been some challenges and issues with regards to the above-mentioned methods.

During static analysis, certain reset signal classifications led to false positives, requiring manual inspection to validate their interpretation. From a design perspective, static security violation analysis required contextual understanding to determine whether they were actual risks or benign patterns. Initial evaluations revealed challenges in analyzing designs with extensive parameterization or macro usage, particularly when struct and pragma constructs were involved.

The asset-centric approach had challenges of its own, such as in the case of the pre-analysis phase that involved scoping, asset identification, threat development, and assessing the existing design in terms of countermeasures, accessible paths, etc. On top of this, while this method helped find violations, it required high involvement of verification engineers and even designers to corroborate the results and initial brainstorming in the pre-analysis phase. Also, the effectiveness of this method was heavily dependent on the quality of functional test cases. Inadequate stimulus coverage would lead to a degraded performance on the analysis.

Side-channel analysis was effective for basic AES implementations, though its applicability to advanced modes and broader cryptographic engines remains a topic for future exploration. Further, it did not give any indications that it can handle multiple engines at the same time, and other non-AES engines. In addition, detecting meaningful leakage requires a large number of traces, which obviously increases simulation and analysis time.

Some general observations across various security methodologies in this framework were that security verification demanded a different approach from functional verification, where the engineers had to think with regard to assets, threat models, and unwanted behaviors in unexpected scenarios. Further, it required discussions with designers both before and after the analysis. Although initial setup may require effort, a well-defined strategy and template for one design block can often be adapted for similar contexts. This approach may apply to publicly available designs like Caliptra, where reusable security verification templates can be adapted based on design context.

VII. Future work

As we have highlighted several challenges above, these certainly present opportunities for improvement in the future. While we cannot disclose details of internal design evaluations or name third-party tool suppliers, we are actively engaged with multiple vendors. A key goal is to establish a robust static security linting methodology as the first line of defense while we continue to improve dynamic security verification methods. Regarding the upfront effort required for some security verification methods, it is best to involve architects, designers, and the verification team as early as possible to ensure awareness and alignment before it becomes too late.

VIII. Conclusion

This paper clearly highlighted the efficiency of security verification methodologies that were used across various designs and IPs. While hardware security verification is not as mature as functional verification and is continuously evolving, several companies have started to embrace it. Unfortunately, there are not many open discussions in the industry that will help set some standard practices and make security verification much more effective, whether by using internal methods or third-party tools. This paper also underscores some gaps in current methodologies that will aid in the improvement of security verification methodologies in the future.

References

[1] M. A. Alfaro Zapata, A. Shahshahani, and Z. Zilic, “Automated Assertion Checker Generator and Information Flow Tracking for Security Verification,” in Proc. 2024 25th Int. Symp. on Quality Electronic Design (ISQED), San Francisco, CA, USA, Apr. 2024, Doi: 10.1109/ISQED60706.2024.10528770.

[2] MITRE CWE, “CWE-1254: Example Weakness Classification,” MITRE, [online]. Available: https://cwe.mitre.org/data/definitions/1254.html

[3] F.-X. Standaert, “Introduction to Side-Channel Attacks,” in Secure Integrated Circuits and Systems, Springer, 2010.

Appendix

Acronyms

SoC	System on Chip
FSM	Finite State Machine
CWE	Common Weakness Enumeration
RTL	Register Transfer Level
IFT	Information Flow Tracking
SCA	Side Channel Analysis
NVMe	Non-volatile Memory express
AXI	Advanced eXtensible Interface
AES	Advanced Encryption Standard
eHSM	enhanced Hardware Security Module

Disclaimer

While absolute security cannot be guaranteed, the authors have evaluated the proposed methodologies on representative designs to validate their practical applicability. These evaluations were conducted in controlled environments to demonstrate the ability of the techniques to detect known issues and provide consistent results. The intent is to illustrate the effectiveness of the security verification framework and scope for improvement in future.