SoC Security Properties and Rules

Nusrat Farzana, Farimah Farahmandi, and Mark Tehranipoor
Florida Institute for Cybersecurity (FICS) Research
Department of Electrical and Computer Engineering,
University of Florida, Gainesville, Florida
Email:{ndipu}@ufl.edu, {farimah, tehranipoor}@ece.ufl.edu

***Abstract*—A system-on-chip (SoC) security can be weakened by exploiting the potential vulnerabilities of the intellectual property (IP) cores used to implement the design and interaction among the IPs. These vulnerabilities not only increase the security verification effort but also can increase design complexity and time-to-market. The design and verification engineers should be knowledgeable about potential vulnerabilities and threat models at the early SoC design life cycle to protect their designs from potential attacks. However, currently, there is no publicly available repository that can be used as a base to develop such knowledge in practice. In this paper, we develop 'SoC Security Property/Rule Database' and make it available publicly to all researchers to facilitate and extend security verification effort to address this need. The database gathers a comprehensive security vulnerability and property list. It also provides all the corresponding design behavior that should be held in the design to ensure such vulnerabilities do not exist. The database contains 67 different vulnerability scenarios for which 105 corresponding security properties have been developed till now. This paper reviews the existing database and presents the methodologies we used to gather vulnerabilities and develop such comprehensive security properties. Additionally, this paper discusses the challenges for security verification and the utilization of this database to overcome the research challenges.**

***Index Terms*—Vulnerabilities, Threat Models, Security Properties, Verification, Property Package, Assertion.**

I. Introduction

The evolution of modern computing systems largely depends on the processing capability of System-on-Chip (SoC). A SoC contains unique functional blocks, known as intellectual property (IP) cores, from different domains (digital, analog, memory, re-configurable fabric, etc.). Moreover, the IP cores participate in complex operations, communications, and co-ordinations with each other to deliver the SoC's expected functionality. These IP cores may bring in unique security challenges due to vulnerabilities associated with their characteristics and interaction with each other when deployed under the real world workload [1]. Therefore, the security of SoCs must be verified at the design stage before fabrication and deployment.

The introduction of security vulnerabilities at different stages of the SoC design flow makes the designing of a secure SoC more challenging. Some of these security vulnerabilities can be introduced unintentionally by a designer during the transformation of a design from specification to implementation. In addition, computer-aided design (CAD) tools can unintentionally introduce additional vulnerabilities in SoCs [2]. Moreover, rogue insiders in design houses can perform malicious design modifications that create a backdoor to the design. An attacker can utilize the backdoor to leak critical information, alter functionalities, and control the system. In general, design houses spend most of their design and verification efforts to meet different criteria (e.g., the correctness of functionality, the optimization of an area, power, and performance, the various use cases, etc.) [3]–[6]. However, they are seldom mindful of potential security threats during design time.

SoC vulnerabilities can be divided into several classes (e.g., information leakage, access control violation, side-channel/covert channel leakage, presence of malicious functions, exploitation of test and debug structure, and fault-injection attacks) [2], [7]–[9]. As SoC complexity continues to grow and time-to-market shrinks, verification has become even more difficult to ensure that these vulnerabilities do not exist in a design. However, a set of well-defined security properties can help verification engineers alleviate the problem.

In the context of security verification and validation, a security property is a statement that can check assumptions, conditions, and expected behaviors of a design [10]. The security property formally describes the expected behaviors of the design for a specific security vulnerability. The coverage of security properties can be used as a metric for the security assessment of a SoC.

We propose a database of security properties to support the verification effort at the pre-silicon design stages. The security properties in the Security Property/Rule Database feature various IP modules that facilitate vulnerability detection in challenging scenarios. By utilizing different open-source design implementations, we check the developed security properties. The rest of the paper is organized as follows: Section II reviews the existing property-based verification and discusses challenges. Section III describes the proposed database in detail. Section IV introduces the experiment methodologies, and how the security properties will be utilized are presented in Section V. Finally, the conclusion and future work are summarized in Section VI.

II. Background

A. SoC Security Verification using Property Checking

The verification engineers utilize a property-driven verification approach for ensuring the correctness of a design. The violation of a property implies that the design should be fixed. As shown in Figure 1, a formal design model is verified against a set of properties, represented as an assertion/cover statement using a model checker tool. The model checker tool provides counter-examples if the property is not held in the design and helps a designer diagnose their design. These properties are used to find the sources of functionality violation in simulation-based verification [11]. Moreover, the synthesizable properties are placed on silicon as a monitor for specific events at the run-time or provide closures for post-silicon validation [12]. However, such functional properties are not sufficient to verify the security and trustworthiness of a SoC. There are some fundamental differences between the functional and security verification approaches. Therefore, we need to develop security properties for SoC security verification and validation, which may or may not overlap with the functional properties.

Today's CAD tools are not well-equipped with the understanding of the security vulnerabilities [13]. Therefore, during logic synthesis, placement, clock-tree synthesis, and routing, CAD tools perform design flattening and multiple optimization processes. They could potentially merge trusted blocks with the untrusted ones, as shown in Figure 2. Though the merging of blocks is not functionally incorrect, a potential information leakage may happen between the blocks due to a structural path between them. Hence property-driven functional verification may not be able to identify such security vulnerabilities.

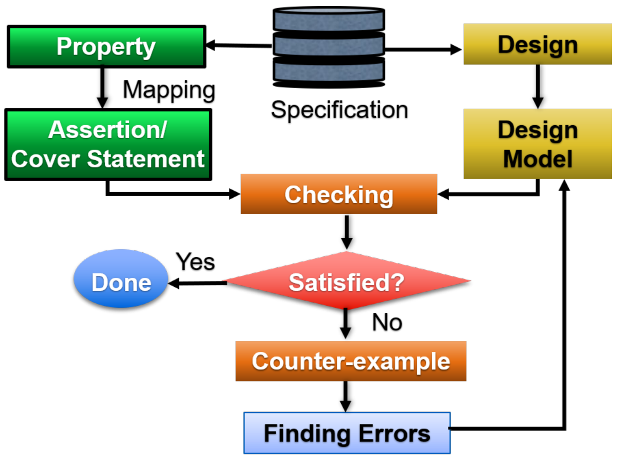


Fig. 1: Property driven verification approach.

B. Existing Methods

Although property-driven security verification has become a bare necessity, only a few works have proposed security properties/policies for modern SoCs. Zhang et al. [14] presented a methodology to infer security-critical properties from known processor errata. However, manufacturers only release errata documents enumerating known bugs. Unknown security vulnerabilities can exist in a SoC design that are not listed in available errata documents. The authors in [15]–[19] utilize information flow tracking (IFT) and statistical models to map hardware security properties. Nevertheless, this technique requires design instrumentation and tainting all the input variables, which require more computational time and memory resources. Hence, IFT and statistical modeling become more complex with increasing design complexity.

Nahiyan et al. [2], [13] proposed a technique to analyze the vulnerabilities of FSM, called AVFSM. Apart from the FSM, a SoC may contain other modules (exception handlers, test and debug infrastructures, random number generators, etc.) which must be inspected during security validation. Authors in [20] provided a method to write the security-critical properties of the processor. They found that the quality of security properties is as good as the developer's knowledge and experience.

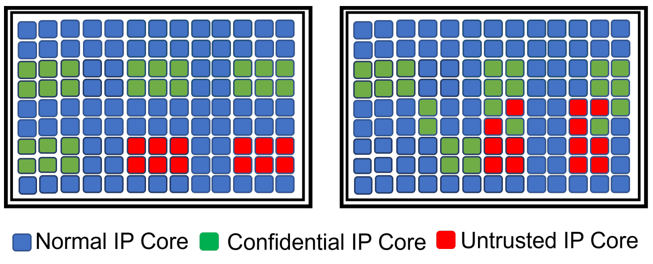


Fig. 2: An example of a synthesis tool's 'melting' different IP cores into one by circuit flattening.

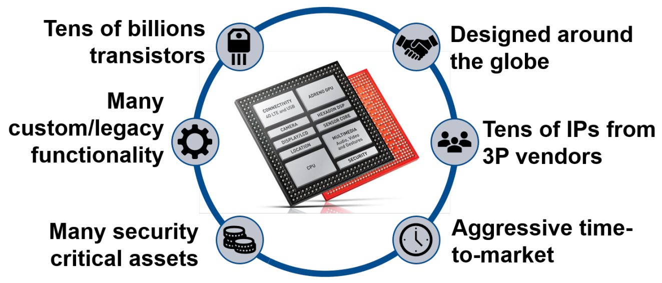
Moreover, there is a lack of threat model definition, which must be considered while developing security properties. There are some other approaches which have developed security properties and metrics by considering only a small subset of vulnerabilities (e.g., the vulnerability in hardware crypto-systems [21], side-channel vulnerabilities [22], [23] and abstraction level limitations like the behavioral model [24]).

In this context, we believe a more effective approach would be to create a database of security properties and map them into assertions to verify them using appropriate tool sets [25]. While developing the database, we have considered a broad spectrum of security vulnerabilities for different entities and threat models.

C. SoC Security Challenges

A large number of hardware vulnerabilities and emerging attacks have been reported over the past decades [2], [7], making the assumption of hardware being secure and trustworthy no longer valid. Due to the semiconductor design process and manufacturing globalization, However, the impacts of hardware vulnerabilities are much more profound than software vulnerabilities, as opportunities for a quick response to hardware threats are minimal. While hardware vulnerabilities are fewer than software vulnerabilities, the detection, mitigation, and responsive actions are far more challenging to put in place for hardware vulnerabilities [26]. It is often quite impossible to change or update a hardware design after deployment into the field. Therefore, it is mandatory to check if any security vulnerability exists in the pre-silicon design stages. However, writing security properties for all possible attack scenarios needs comprehensive knowledge and experience of the subject matter expert.

Figure 3 depicts the major factors that make security assurance of a SoC more challenging. The design and implementation of a SoC is not in-house anymore and have spread around the globe. Different IPs are coming from third-party IP vendors in soft, firm, and hard IP forms. Providers do not ensure the security of the


Tens of billions transistors
Designed around the globe
Many custom/legacy functionality
Tens of IPs from 3P vendors
Many security critical assets
Aggressive time-to-market
Fig. 3: SoC security challenges.

custom/legacy IPs. Since SoCs can perform difficult operations, IP cores come from different domains (analog, digital, reconfigurable, fabric, etc.). Security verification of each IP core, along with cross-domain checking, is required for security assurance of SoCs. Moreover, as the number of transistors increase, so does the complexity of a SoC. Yet, the time-to-market of a SoC is shrinking. Design houses spend optimal time testing a SoC, however, security verification is not a significant concern for most of them. Hence, in the competitive market of a SoC, security is often neglected. Along with these issues, most design engineers are not concerned about the critical security assets in their design. Since assets can be diverse and vulnerable to many attacks, a designer must decide which type of assets need protection against possible attacks. Choosing the wrong asset against an unsuitable set of attacks can waste time and resources while leaving the design susceptible [27].

III. Description of Database

The intention to develop the security property database is to guide design and verification engineers to identify vulnerabilities at the very beginning of the SoC design cycle. The database is designed to perform formal verification of designs in RTL, Gate Level, and Physical Layout Level. At first, the security-critical properties were developed in natural language for different entities, respective vulnerabilities, and threat models. Then the security properties were formally represented to verify different design implementations. All security properties with detailed descriptions and demonstrations have been wrapped into readily available packages for the users. We have elaborately described the database in this section.

A. Identification of Security Properties

We have identified the security properties for SoCs by following several steps as described in Algorithm 1. The algorithm takes a design $\mathbb{D}$ as input and identifies a set of security-critical properties $\mathbb{P}$ as output by going through the following steps: 1) asset identification, 2) vulnerability identification for identified assets and vulnerability database creation, 3) threat model development, and 4) property formulation as described in this section.

1) Step 1: Choice of Assets: To develop security properties, one must first understand the underlying assets and their corresponding security levels. The choice and specification of an asset can be different from one abstraction level to another, in addition to the varied adversarial capabilities and intents.

Algorithm 1: Algorithm for Security Critical Property Identification

```
    Input: Design of an SoC/ IP, $\mathbb{D}$
    Output: Set of security properties, $\mathbb{P}$
1   /*Step1 : Identify Primary and Secondary Assets*/
2   $\mathbb{A}$ = Identify_Assets($\mathbb{D}$);
3   for Asset $A \in \mathbb{A}$ do
4       if $A \in$ Primary_Assets then
5           $\mathbb{A} = \mathbb{A} \cup$ Identify_Secondary_Assets(A);
6       end
7   end
8   $\mathbb{P} = \{\}$;
9   $\mathbb{AL}$ = Abstraction_levels(D);
10  for Asset $A \in \mathbb{A}$ do
11      /*Step 2: Identify Possible Vulnerabilities and Associated Abstraction Levels*/
12      for Various Abstraction levels $AL \in \mathbb{AL}$ do
13          $\mathbb{V}_{\mathbb{AL}}$ = Identify_Vulnerabilities $(D, AL, A)$
14          for Vulnerability $V \in \mathbb{V}_{\mathbb{AL}}$ do
15              /*Step 3: Develop Threat Models*/
16              T = Identify_Threat_Model $(A, V, AL)$;
17              /*Step 4: Property Formulation*/
18              $\mathbb{P} = \mathbb{P} \cup$ Formulate_Property$(D, A, AL, V, T)$;
19          end
20      end
21  end
22  return $\mathbb{P}$
```

An asset can be defined as a resource with a security-critical value that is worth protecting from the adversaries [28]. Assets in an SoC can be classified as: 1) *Primary Assets* - those that are the ultimate target for protection as deemed by the designer. For example, primary assets include device keys and protected data (e.g. a password), firmware, device configuration, communication credentials, and entropy for a true random number generator (TRNG), and physically unclonable functions (PUFs). 2) *Secondary Assets* infrastructures that require protection to ensure the security of the primary assets during rest or transition. A shared bus is an excellent example of a secondary asset as its protection ensures the security of a key moving from an on-chip ROM to an encryption engine within a SoC. Moreover, the asset under consideration can be static or dynamic, based on how it is generated and utilized. The classification of assets are depicted in Figure 4.

As shown in Algorithm 1 (lines 3-7), a set of assets $\mathbb{A}$ will be available at the end of this step by analyzing the design $\mathbb{D}$. After choosing the assets, the designer must analyze their propagation paths to find potential vulnerabilities and weaknesses along with their access points.

2) Step 2: Development of Vulnerability Database: As discussed earlier, unintentional vulnerabilities can arise at any design stage in addition to intentional malicious modification and exploitation. Therefore, the next step should be finding vulnerabilities to access the assets to perform exploitation.

By surveying the available literature, open-source vulnerability databases (e.g., Common Weakness Enumeration, Common Vulnerability Enumeration), design specifications, common design mistakes, and studying the implementation of the design in detail, an attacker's points of entry or potential vulnerabilities can be addressed. Eventually, a database of potential vulnerabilities $\mathbb{V}$ can be established based on the design and the identified assets (as shown in line 11 of Algorithm 1). For example, it has been shown that the AES key can be extracted using different attack strategies such as power and timing side-channel attacks [29]–[32], trace buffer and scan attacks [33], [34], Hardware Trojan insertion [35], [36], physical attacks [37], and more that have been developed over time. With existing attack information, we can point out essential vulnerabilities of a weak AES implementation [2], like an attacker's access to FSM's protected state, or one's ability to the key through plain-text or control signals, etc. [38]. Identifying these intentional and unintentional vulnerabilities of a design is essential to define the threat models and develop the corresponding security properties.

3) Step 3: Development of Threat Models: After steps 1 and 2, with the knowledge of design implementation, assets to protect, and identified vulnerabilities, it is easier to develop threat models $\mathbb{T}$ for each asset (as shown in line 16 of Algorithm 1). Examples include deviation from the expected functionality, illegal access path, and

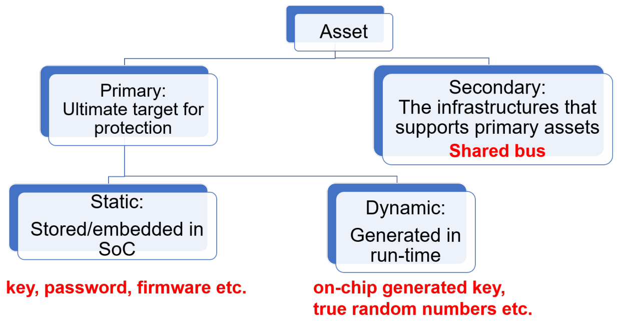

Asset
Primary: Ultimate target for protection
Secondary: The infrastructures that supports primary assets
Shared bus
Static: Stored/embedded in SoC
Dynamic: Generated in run-time
key, password, firmware etc.
on-chip generated key, true random numbers etc.

Fig. 4: Classification of security assets.

corrupting the critical data and FSM states. For the database, we have classified threat models into three categories:

- *Confidentiality Violation:* The unauthorized disclosure of confidential/secret information [39], [40]. Examples of confidentiality violation include the flow of assets to an untrusted IP or to observable points, insertion of a Hardware Trojan [41], [42] to the unspecified functionality of the design to create information flow, any side-channel vulnerability that provides information about the timing, power consumption, or electromagnetic emanation of an operation to the attacker - are examples of confidentiality violations. An example of a security property to check confidentiality violation is- "Design assets should not be propagated to and leaked through any observable point, such as primary outputs, DFT, or DFD infrastructures."
- *Integrity Violation:* An integrity violation means that confidential data can be written or modified by an entity who is not authorized to do so [40], [43]. Unauthorized interaction between trusted and untrusted IPs, illegal access to protected memory addresses/ protected states of a controller module, and an out-of-bound memory access fall under this threat model. An example of a security property to check integrity violation is- "A security-critical entity should be isolated from entities/input ports that are not authorized to control and/or modify it"
- *Availability Violation*: Disruption in the service or connectivity of the modules within a SoC can be categorized as an availability violation threat. Numeric exceptions (e.g. divide by zero), the introduction of a delay in a regular operation, and deadlock creation to make a resource unavailable, can be considered examples of service disruption. Therefore, an adversary can utilize the extra time or malfunction for leaking information or violating access controls. A sample security-critical property could be- "Security-critical entities should be isolated from other entities, or memory access should be restricted during unexpected delays. "

A single vulnerability can fall under the umbrella of multiple threat models $\mathbb{T}$ for a single asset A. For example, adversarial access to the intermediate results of encryption performed by an AES module through debug port can be categorized as confidentiality and integrity violation. Properties should be written to check against all vulnerabilities and threats for each asset under consideration.

4) Step 4: Identification of Properties: For a design under consideration, the information about an asset, corresponding vulnerabilities, and threat models assists (as shown in line 18 of Algorithm 1) to formulate security properties $\mathbb{P}$. For a specific entity, a single threat model/vulnerability can be mapped into multiple properties or vice-versa. The security properties are written so that they negate the vulnerability scenario. If a design implementation holds the behavior described by the security property, it is considered secured otherwise insecure.

B. Categories of Security Properties

Since different vulnerability scenarios can get introduced at different stages of a SoC's life cycle, we need to consider the time at which the property should be verified. Some properties should be checked statically at the design time. For example, the registers containing the critical status of a CPU should be accessed only in valid ways, as any undefined access to these registers is considered a threat. On the other hand, during boot time/reset time, intermediate buffers of the design may be initialized to "don't care" values. There should be properties that ensure that those "don't care" values in boot time do not lead to vulnerabilities that provide attackers the authority to start from a security-critical state of the design and steal some critical information or cause the denial of service. As some assets would be generated at run-time, we require security-critical properties in the form of sensors or monitors [44] that can operate in run-time to prevent malicious functionality, denial of service, etc.

Thus, the properties obtained from the framework can be classified into 1) static/design-time, 2) boot-time, and 3) run-time. To check the static/boot-time properties, design D can be translated into formal models [44], and properties are mapped to LTL/CTL formulas [45]. Model checker tools can be used to verify formal properties at the design/boot time. However, to check run-time properties, design D needs to be instrumented with IFT capabilities. Simulation-based formal verification tools can be used to verify the properties at run time.

C. Conversion of Security Properties to Assertions:

As the developed security properties are in natural language, users need to convert them into assertions/cover statements to make them readable for verification tools. An assertion is a check embedded in a design or bound to a design. A single bit associated with an assertion can indicate the pass or fail status if the design does not hold the property. On the other hand, a cover statement can check if a specific scenario has ever occurred in the design during the simulation. Therefore, cover statements are mostly used as coverage information for design validation. For example, suppose the assertion is not covered during the run-time. In that case, an assertion is triggered at the end of the execution of the cover statement.

For a specific design example, a user must identify appropriate signals and concatenate them using necessary operators based on the associated security property. Thus, the security property is converted into an assertion/cover statement. Multiple assertion/cover statements may need to be written for a single property. Moreover, the same property can be converted into different assertions for different design implementations and different abstraction levels.

D. Organisation of Property Database

We create a database using the security properties we have formulated following the method as described in Section-III-A. A sample database has been depicted in Table-II and Table-III. The database consists of the following columns:

1) **Abstraction Level:** Each security property is associated with one or more abstraction levels. Since a new vulnerability can be introduced while translating a design from one abstraction level to another, new security properties may also need to be developed. Therefore, this column provides the user of the database the information about the associated abstraction level.
2) **Family of Entity:** In this column, we have enlisted some commonly used IP modules (e.g., debug and HALT unit, AES module, trace buffers, exception handler, TRNG, current program status register, bus protocols, etc.) for which we developed security properties by following the steps described previously.
3) **Vulnerability Source:** Before providing the detailed description of the vulnerability under consideration, we enlist the primary reason for which the vulnerability may occur in this column.
4) **Overlapping with CWE or not:** Common Weakness Enumeration (CWE) is a widely used database where almost 117 hardware weaknesses are enlisted [46]. Since weaknesses are errors that can lead to vulnerabilities, the database can be used as a resource to find different vulnerabilities. Apart from the CWE database, we enumerated new vulnerability scenarios. We use this column to keep track if a vulnerability scenario is common with CWE or not.
5) **Vulnerability Description:** In this column, we provide a detailed description of the vulnerability under consideration (e.g., how a vulnerability source is exploited to perform an attack).
6) **Associated Assets:** A single vulnerability can be associated with multiple assets. Hence, security properties should be developed for each of the associated assets.
7) **Threat Model:** A single vulnerability can be associated with multiple threat models as described in Section-III-A. For each threat model, security properties should be identified.
8) **Property:** Identified security properties which negate the corresponding vulnerability scenario are enumerated in this column.
9) **Time to Check:** Based on the vulnerability, the property may need to be checked in static time, run-time, or boot-time. This information becomes helpful while converting the properties into assertions.
10) **Design-agnostic or not:** This column helps the user to identify if a specific property is applicable for any design implementation or if one should use it for an architecture-specific design.
11) **Benchmark:** This column describes the design example for which the corresponding property is checked. Mostly, we used open-source design examples that are publicly available and widely used for simplicity.
12) **No. of Assertions:** The properties are converted into assertions for the benchmark in column 11. A single property may need multiple assertions to describe the intended design behavior.
13) **Can be checked or not:** In some cases, even though we develop security properties, it becomes very challenging to check the properties due to design complexity, the user's inability to find appropriate design signals, and/or hardware-software interactions, etc.
14) **Tool Used for Checking:** We enlist the industry standard and publicly available verification tool that we use to check a security property. The information helps the user get an idea of how to translate the properties into assertion using a different verification tool.
15) **Counter-example:** If the assertions are violated,

the verification tools provide violation traces. The traces are useful for the designers to fix the incorrectness in the design.

16) **Property name:** We have named each of the properties we have developed using the associated abstraction level, entity name, and vulnerability. A detailed description of the naming convention is provided in Section-III-F. It will help the user to identify a set of properties they want to look into.

E. Structure of Property Database

To make the security property database readily available to the user, we have converted each row of the database into packages. The packages are designed to perform formal verification of designs in RTL, Gate level, and Physical layout level. As the properties are derived from different vulnerability sources, each having its specific format, we must create the database efficiently. Each package contains design implementation source code, corresponding properties written in the form of assertions, a script to verify the assertions using formal verification tools, and counterexamples (if provided by the verification tool). If the associated abstraction level is gate-level or physical layout level, then the package contains a technology file as depicted in Figure 5.

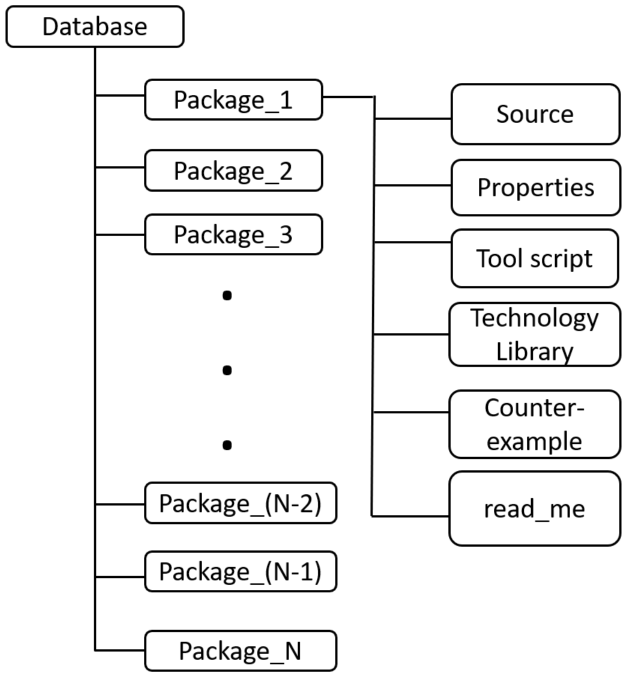


Fig. 5: Structure of security property database.

F. Naming Convention of the Property Packages

We have developed packages of different entities and corresponding vulnerabilities. Moreover, a property package can be associated with a specific abstraction level. Furthermore, it is possible that new security vulnerabilities can come up, so the database must be updated with new property packages over time. Based on the above, we develop the convention as shown in Figure 6 to assign a unique name to each property package:

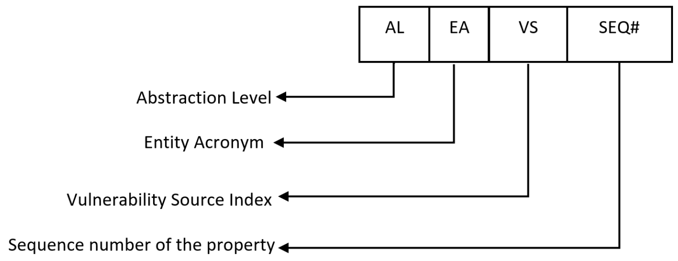


Fig. 6: Naming convention of property packages.

- **Abstraction Level:** The first letter of the abstraction level associated with the property. For example RTL $\rightarrow$ **R**, Gate level $\rightarrow$ **G**, Physical Layout level $\rightarrow$ **P**.
- **Entity Acronym:** The initials of the family of entity are used to fill the second field of the package name. Examples are shown in Table-I.
- **Vulnerability Source Index:** Vulnerabilities have been indexed with a two-digit number for each family of entities at a specific abstraction level as depicted in Table-I.
- **Sequence Number of the Property:** All the properties under the hood of the same vulnerability are chronologically ordered with sequence numbers.

IV. Experimental Result

This section describes some sample security properties that we have developed by surveying the literature, available open-source architectures, design implementations, and different attack strategies. We also demonstrate how we convert the security properties into packages for suitable design examples.

Sample Security Properties: We have considered the Test and Debug structures as an example. Features used for test and debug units include reading and modifying register values of processors and peripherals. Usually, debugging involves halting the execution once a failure has been observed and collects state information retrospectively to investigate the problem [47]. During HALT operation, a processor stops executing according to the instruction indicated by the program counter's (PC) value, and a debugger can examine and modify the processor state via Debug Access Port (DAP). On

TABLE I: DETAILS OF NAMING CONVENTION OF PROPERTY PACKAGES

Family of Entity	Entity Acronym	Abs. Level	Vulnerability Source	Vulnerability Source index
Test & Debug Structure	TD	Register Transfer Level (R)/ Gate Level (G)	Halt Mode Debugging	01
			Switching between Test mode and Functional mode	02
			Trace Buffer	03
		Register Transfer Level (R)	Inter-processor Debugging	04
			Core-site External Debug Registers	05
			Debug interfaces & Debug registers (Breakpoint Exception (BP) and Debug Exception (DB))	06
			Debug initialization	07
Encryption Module	EM	Register Transfer Level (R)	Output ports of AES	01
			Weakness in AES implementation	02
			Output ports of RSA	03
			Weakness in RSA implementation	04
			Weakness in SHA implementation	05
			Too small modulus of RSA encryption	06
			Lack of entropy in key pair generation of RSA	07
			Low private exponent of RSA encryption	08
			Observable discrepancy	09
		Gate Level (G)	Output ports of encryption module	01
			Weakness in encryption algorithm implementation	02
			Accessibility of debug units to intermediate registers	03
			Timing information of computation	04
			FSM State encoding	05
			Intermediate states of encryption algorithm	06
True Random Number Generator (TRNG)	TG	Register Transfer Level (R)	Harvesting Mechanism of TRNG module	01
			Biasness in TRNG module	02
Controller Design	CD	Register Transfer Level (R)	Security critical FSM of controller circuit	01
			Content of current program status register (CPSR)	02
			Memory Management Unit (MMU)	03
			Load-store Unite (LSU)	04
Exception Handler	EH	Register Transfer Level (R)	Position of Exception vector table	01
			Mode switching	02
			Exception register value updating	03
Hardware IP interconnect	HI	Register Transfer Level(R)	Incorrect Synchronization between master and slave IP modules	01
Locking mechanism	LI	Register Transfer Level (R)	Improper implementation of lock protection to registers	01
			Improper lock behavior after power state transition	02
			Debug mode	03
Core and Computation Issue	CCI	Register Transfer Level (R)	CPU configuration to support exclusivity of write and execute operations	01
			Instruction set architecture	02
Memory and storage issues	MSI	Register Transfer Level (R)	Improper write handling in limited -write nonvolatile memory	01
			Mirrored regions with different values	02
			Improper access control applied to mirrored or aliased memory regions	03
Miscellaneous	MI	Register Transfer Level (R)	Incorrect signal connection	01
			Reserve bits	02
			Instantiation of register module	03
			Write once register	04
			Illegal or undocumented instruction execution	05
			Erroneous reset configuration	06
			Handling interrupt request	07

TABLE II: SAMPLE SECURITY PROPERTY DATABASE

Abs. Level	Family of Entity	Vul. Source	Does it overlap with CWE?	Vulnerability Description	Associated Assets	Threat Model	Property	Time -to-check	Property Name
Register Transfer Level (R)	Test & Debug Structure (TD)	Halt mode debugging (01)	No	Readability of/ writability to sensitive information through Debug Access Port (DAP) during HALT operation	Program counter's value	Integrity violation	Program counter's value should not be modified during HALT operation (1)	Static	RTD011
						Confidentiality violation	Program Counter's value should not be readable through Debug Access Port (DAP) when a processor is not in HALT mode debugging (2)	Static	RTD012
					Privilege Memory	Integrity violation	Readability to privileged memory should be denied in HALT operation through DAP (3)	Static	RTD013
						Confidentiality violation	Writability to privileged memory should be denied in HALT operation trough DAP (4)	Static	RTD014
Gate Level (G)	Test & Debug Structure (TD)	Switching between Test mode and Functional mode (02)	No	By switching the chip between functional mode and test mode, secret information can be scanned out or calculated. Hence, the design can be prone to scan-based attack.	Crypto key, password, user information	Confidentiality violation	A reset signal should be enabled for the scan flip-flops whenever there is a switch from the functional mode to the test mode (2)	Static	GTD022
	Encryption Module (EM)	Accessibility of debug units to intermediate registers (03)	No	Observability of encryption key from the test and debug ports can help attackers to leak information about the AES key	Crypto key	Integrity violation	Key registers should not be observable from test and debug ports. (1)	Static	GEM031
				Controllability /observability of the FSM state flip flops from test and debug ports can allow an attacker to bypass intermediate rounds of encryption.	Crypto key	Confidentiality violation	FSM State flip-flop should not be controllable and observable from the additional test and debug ports. (2)	Static	GEM032
		Intermediate states of encryption algorithm (06)	No	Because of the synthesis tool optimization, the protected state can be accessed from an unauthorized state. (e.g. don't care states)	Protected States of FSM of encryption algorithm	Confidentiality violation Integrity violation	FSM encoding of encryption algorithm should be implemented in such a way that an FSM state cannot be accessed from a don't care state to resist fault injection vulnerability. (1)	Static	GEM061

TABLE III: VERIFICATION RESULTS OBTAINED FROM ASSERTION CHECKING

Property Name	Design Agnostic?	Can be checked or not?	Benchmark	Assertion	Tool Used	Status
RTD011	Yes	Yes	MSP430 micro-controller	check_spv -create -from {dbg_uart_rxd} -from_precond {dbg_0.dbg_en_s==1'b1} -to {frontend_0.pc} -name "receiver_to_pc"	JasperGold SPV	Violated
RTD012	Yes	Yes	MSP430 micro-controller	check_spv -create -from {frontend_0.pc} -from_precond {dbg_0.dbg_en_s==1'b0} -to {dbg_uart_txd} -name "pc_to_trans"	JasperGold SPV	Violated
RTD013	Yes	Yes	MSP430 micro-controller	check_spv -create -from {mem_backbone_0.dmem_dout} -from_precond {dbg_0.dbg_en_s==1'b1} -to {dbg_uart_txd} -name "memory_to_transmitter"	JasperGold SPV	Violated
RTD014	Yes	Yes	MSP430 micro-controller	check_spv -create -from {dbg_uart_rxd} -from_precond {dbg_0.dbg_en_s==1'b1} -to {mem_backbone_0.dmem_din} -name "receiver_to_memory"	JasperGold SPV	Violated
GTD022	Yes	Yes	RSA core	property testmode; @(posedge clk) (RSA_binary.test_se==1) ##1 (RSA_binary.test_se==0) |->(RSA_binary.RESET==1); endproperty	JasperGold	Violated
GEM031	Yes	Yes	An AES implementation	check_spv -create -from {key} -to {test_so} -name "observability_of_key"	JasperGold SPV	Violated
GEM032	Yes	Yes	An AES implementation	//controllability checking check_spv -create -from {test_si} -to {next_FSM} -name "controllability_of_state_flip_flop" //observability checking check_spv -create -from {next_FSM} -to {test_so} -name "observability _of_state_flip-flop"	JasperGold SPV	Violated
GEM061	Yes	Yes	An AES implementation	property finalRoundAccessFromDontCare; @(posedge clk) (aes_binary.FSM==FINAL_ROUND) |-> $past (aes_binary.FSM==3'b111) | $past (aes_binary_FSM==3'b001) | $past(aes_binary_FSM==3'b101) endproperty	JasperGold	Violated

the other hand, Design for testing (DFT) adds features to make it easier to develop and apply manufacturing tests to the designed hardware, thus giving access to the intermediate registers of the design.

As the user has read/write access to memory and CPU registers in the debug mode, several threat models can be developed [48]. As shown in Table II, (first four rows), we identified assets of an SoC with debugging features and their vulnerabilities first. Then we enlisted probable threat models of the vulnerabilities. For example, when we consider the PC's value as an asset, we will need to check whether it is accessible from the debug access port or not. If so, an attacker can freeze the system and modify the PC's value so that the sequence of code execution can get redirected after a HALT operation is executed. Similarly, when the asset is in a particular memory address, writing to a protected memory address from DAP can result in firmware modification or malicious code injection threats. Moreover, when the asset is in a particular memory address, reading from that address during HALT operation may leak sensitive information (e.g., password, key, etc.). Here, the first two scenarios are integrity violations. In contrast, the last two are examples of confidentiality violations resulting from a single vulnerability source (HALT mode debugging). We have derived the security properties RTD011, RTD012, RTD013, RTD014 for the vulnerabilities mentioned above.

Moreover, when an RTL design of a SoC is synthesized to gate-level with DFT insertion, all flip-flops in the design are replaced with scan flip-flops (SFF). The scan flip-flops are connected to one or more shift registers in the test mode, making them controllable/observable from primary input/output. Most state-of-the-art scan-based attacks rely on the vulnerability of switching from the functional mode to the test mode [49]. The attackers assume that the data in the scan flip-flops can be preserved during mode switching. Therefore, the development of security property GTD022 implies that a reset signal should be asserted while switching between functional and test mode.

Another category of vulnerability is related to the enhanced controllability/observability of a design due to on-chip Design for Testing (DFT) instrumentation. The selected signals to be controlled and observed by DFT instrumentation are usually related to important system states. Attackers can manipulate the control and data flow (integrity violation) at their will to reach system states that might leak confidential information. For example, controllability/observability of Finite State Machine (FSM) through test and debug ports can allow an attacker to perform a fault injection attack and bypass the intermediate rounds of encryption [2], [50]. In addition, observability of encryption key registers from the test and debug ports can help attackers to leak information about the key (confidentiality violation). Hence, security properties GEM031 & GEM032 are evolved to make sure that any flip-flops in the design are not a part of a scan chain or controllable/observable from any test or debug port.

As described earlier, synthesis tools do not deal with security issues; instead, many security vulnerabilities can be unintentionally introduced. To meet the cost and performance constraints, synthesis tools perform optimization during RTL to gate level synthesis. As a result, don't care states may get introduced to the finite state machine (FSM) that attackers can utilize to perform fault injection attacks. Attackers can enter into a protected state from a don't care state by bypassing a security checking. Security property GEM061 implies that any security-critical or protected state should not be accessed from any don't care state.

The scenarios imply that these vulnerabilities do not exist in RTL; however, they can be newly introduced at the gate level.

Sample Security Property Packages:

We mapped the first four properties in Table II into assertions and checked them for the open-source MSP430 micro-controller [47] with the debug unit. Since the properties are developed to negate the debugger's readability/writability to the program counter register and a certain memory region, we considered the receiver and transmitter ports of the debug unit as attackers' entry and exit points, respectively. Then, to check the existence of any functional path between the attacker's entry/exit point to the security asset under consideration, we utilized JasperGold SPV [51] tool. The verification result (shown in Table III) indicates that the properties are violated. Therefore, the underlying vulnerabilities exist in the MSP430 micro-controller.

To check property GTD022, we synthesized an open-source RSA design with DFT insertion. First, we developed an assertion to check if a reset signal is asserted while switching between test and functional modes. Then, we utilized the JasperGold tool to check the assertion. The violation of the assertion implies that an attacker can perform a scan-based attack for the design example.

For checking properties GEM031 & GEM032, we utilized an AES design from Trust-hub [52] and synthesized it with DFT insertion. Then we used an assertion to check if there is any functional path between the test and debug ports to the FSM state register or the key register. The verification result shows that the assertion violates the AES design and that it is vulnerable.

Again, we have checked the security property GEM061 for a synthesized DFT inserted netlist of the AES benchmark. The design under consideration has five FSM states, encoded as WAIT_KEY = 3'b110, WAIT_DATA= 3'b010, INITIAL_ROUND= 3'b011, DO_ROUND= 3'b100, FINAL_ROUND= 3'b000; Therefore, the don't care states are 3'b111, 3'b001, 3'b101. FINAL_ROUND is the most security-critical state where the encrypted message is delivered. Moreover, it should only be accessed from DO_ROUND. The corresponding assertion checks if the FINAL_ROUND state can be accessed from any don't care states or not. The assertion violation shows that there can be a scenario where the FINAL_ROUND is accessed from a don't care state, which can be utilized to perform a fault injection attack.

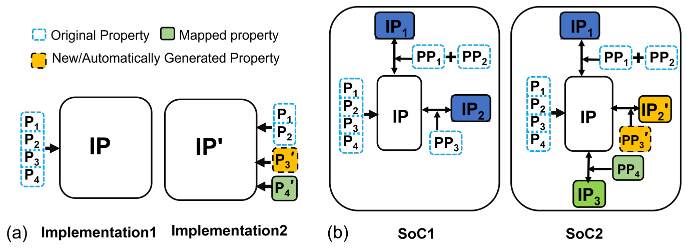
All information in each row in Table-III, are wrapped in zip folders in the form of a property package. The package name is the same as the corresponding property name.

V. Utilization of Property Database

A. *Architecture-agnostic Security Properties*

To reduce the effort of security validation in the SoC design flow, security properties should be designed in architecture-agnostic fashion. Architecture-agnostic security properties can be defined as a set of properties which are universally applicable to a diverse set of devices and implementations rather than being tied to a specific one. Though all the security properties identified for one target implementation cannot be mapped to another, they can be used as a reference for generating new properties that are more appropriate for the latter. If a good percentage of security properties for a certain entity in one implementation can be reusable for the same or similar entity in a different implementation, the effort for security validation will be reduced to a great extent.

Let us consider the example shown in Figure 7(a), where a particular IP has been utilized in Implementation1 & Implementation2. P_1, P_2, P_3, P_4 are the identified security properties for the IP in Implementation1 based on the type of asset it is dealing with, possible vulnerabilities, and threat models. Some of these identified properties can be universal and applied for any


Original Property
Mapped property
New/Automatically Generated Property
IP
IP'
(a) Implementation1 Implementation2
(b) SoC1 SoC2
Fig. 7: (a) Same IP in different implementations (b) Same implementation of an IP in different SoCs.

implementation of the IP. In the example, P_1 & P_2 are the universal properties and remain same for IP′ in Implementation2. However, P_3 & P_4 are the architecture-specific security properties of Implementation1 and can not be reused for Implementation2. Therefore, new security properties P_3' & P_4' will come into consideration for IP′ in Implementation2. The properties which do not match can be generated by tuning different parameters of the implementations or by using the previously identified properties as a reference to generate them automatically to ensure security validation.

For example, the number of rounds can be different for different AES implementations, and security properties can be obtained by modifying the existing properties' timing according to the implementations' specifications. FSM state transition vulnerability would be another good example here. The definition of don't care states are different for different FSM implementations based on the encoding scheme (e.g., gray, binary, one-hot encoding). Security properties for a newer implementation can be identified by changing the definition according to the implementation under consideration. Note that for the new features of the newer implementations of an IP, we need to write a new set of properties to cover them and update the security property database accordingly.

In the case of SoCs, a similar approach can be applied to identify security properties. The analogy has been illustrated in Figure 7(b). The same IP has been integrated into two different SoCs. Since the implementation remains the same for the IP, no new security properties are required. Moreover, some IP interactions (IP $\rightarrow$ IP$_1$), as well as the corresponding security properties PP_1 & PP_2 are indifferent in both SoCs. However, other IP interaction has changed (from IP $\rightarrow$ IP$_2$ to IP $\rightarrow$ IP$_2'$) with the new implementation of IP$_2$ (IP$_2'$) in SoC2 and new security property ($PP_3 \rightarrow PP_3'$) needs to be mapped. Moreover, in SoC2, a new IP module (IP$_3$) is interacting with the IP. Therefore, a new property PP_4 is required for completing the security property set for SoC2. The new security property can be generated automatically by using the previously identified properties of SoC1 as

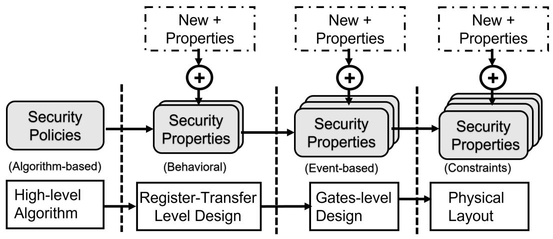


Fig. 8: Security-critical property identification at different abstraction levels.

templates.

B. Property Mapping at Different Abstraction Levels

Even for the same implementation, security properties need to be mapped at different abstraction levels of the SoC design life cycle (high-level specification $\rightarrow$ RTL $\rightarrow$ Gate level $\rightarrow$ Layout) [53], [54]. The security properties identified for high-level specifications can be necessary but not sufficient for security validation in lower levels of abstraction like RTL, gate level, or layout level. Vulnerability definition and asset generation are dynamic as they are tied to different abstraction levels. As described in the example in Section-IV, after insertion of DFT, or trace buffers for testing and debugging purposes, a new set of properties comes into play such as: *Key should NOT leak through the observable test points*. Moreover, RTL to gate-level transformation introduces don't-care states, which require new security properties. As illustrated in Figure 8, we either map the security properties from higher levels of abstractions or introduce new properties to cover new vulnerabilities introduced in lower levels. Thus, security properties will eventually enrich the property database. The database should be updated in the case of new attacks and vulnerabilities. Furthermore, in zero-day attacks, the vulnerability database will help in creating new security properties to identify the underlying weaknesses that facilitate new attacks. In short, the process of security property identification will be a back-and-forth approach. Some properties will be identified manually, while others will be mapped from the previous implementation, and the rest will be generated automatically (using the templates generated from the database) to cover the broad set of security vulnerabilities.

VI. Conclusion

In this paper, we suggested that security should be considered from the very beginning of the SoC design process. The 'Security Property/Rule database' we have developed will help to reduce security validation efforts to a great extent. Furthermore, our developed security properties are flexible for different implementation and resilient against different vulnerabilities. Therefore, along with design specification, synthesis and functional verification, security property identification and verification should be established as another pillar of the hardware design process.

REFERENCES

[1] M. Tehranipoor and C. Wang, *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[2] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, and M. Tehranipoor, "AVFSM: a framework for identifying and mitigating vulnerabilities in FSMs," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[3] P. Mishra, S. Bhunia, and M. Tehranipoor, *Hardware IP security and trust*. Springer, 2017.

[4] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 1–23, 2016.

[5] H. Salmani and M. M. Tehranipoor, "Vulnerability analysis of a circuit layout to hardware trojan insertion," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1214–1225, 2016.

[6] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[7] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE design & test of computers*, vol. 27, no. 1, pp. 10–25, 2010.

[8] G. K. Contreras, A. Nahiyan, S. Bhunia, D. Forte, and M. Tehranipoor, "Security vulnerability analysis of design-for-test exploits for asset protection in SoCs," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 617–622.

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

[10] P. Mishra, M. Tehranipoor, and S. Bhunia, "Security and trust vulnerabilities in third-party ips," in *Hardware IP Security and Trust*. Springer, 2017, pp. 3–14.

[11] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra, "Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution," in *2018 IEEE International Test Conference (ITC)*. IEEE, 2018, pp. 1–10.

[12] F. Farahmandi, R. Morad, A. Ziv, Z. Nevo, and P. Mishra, "Cost-effective analysis of post-silicon functional coverage events," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 392–397.

[13] A. Nahiyan, F. Farahmandi, P. Mishra, D. Forte, and M. Tehranipoor, "Security-aware FSM design flow for identifying and mitigating vulnerabilities to fault attacks," *IEEE Transactions on Computer-aided design of integrated circuits and systems*, vol. 38, no. 6, pp. 1003–1016, 2018.

[14] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying security critical properties for the dynamic verification of a processor," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 541–554, 2017.

[15] W. Hu, A. Althoff, A. Ardeshiricham, and R. Kastner, "Towards property driven hardware security," in *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 2016, pp. 51–56.

[16] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "On the complexity of generating gate level information flow tracking logic," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 3, pp. 1067–1080, 2012.

[17] S. Bhunia and M. Tehranipoor, *Hardware Security: A Hands-on Learning Approach*. Elsevier Science, 2018. [Online]. Available: https://books.google.com/books?id=wIp1DwAAQBAJ

[18] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner, "Leveraging gate-level properties to identify hardware timing channels," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 9, pp. 1288–1301, 2014.

[19] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I2C and USB," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2011, pp. 254–259.

[20] "analyzing hardware security properties of processors through model checking."

[21] B. Yuce, N. F. Ghalaty, and P. Schaumont, "TVVF: Estimating the vulnerability of hardware cryptosystems against timing violation attacks," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015, pp. 72–77.

[22] "side-channel vulnerability factor: A metric for measuring information leakage."

[23] A. Nahiyan, J. Park, M. He, Y. Iskander, F. Farahmandi, D. Forte, and M. Tehranipoor, "SCRIPT: A CAD Framework for Power Side-channel Vulnerability Assessment Using Information Flow Tracking and Pattern Generation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 25, no. 3, pp. 1–27.

[24] H. Salmani and M. Tehranipoor, "Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level," in *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pp. 190–195.

[25] S. Aftabjahani, R. Kastner, M. Tehranipoor, F. Farahmandi, J. Oberg, A. Nordstrom, N. Fern, and A. Althoff, "Special Session: CAD for Hardware Security-Automation is Key to Adoption of Solutions," in *2021 IEEE 39th VLSI Test Symposium (VTS)*. IEEE, 2021, pp. 1–10.

[26] "Hardware vs. software vulnerabilities." [Online]. Available: https://inside.battelle.org/blog-details/hardware-vs.-software-vulnerabilities

[27] N. Farzana, A. Ayalasomayajula, F. Rahman, F. Farahmandi, and M. Tehranipoor, "SAIF: Automated Asset Identification for Security Verification at the Register Transfer Level," in *2021 IEEE 39th VLSI Test Symposium (VTS)*. IEEE, 2021, pp. 1–7.

[28] A. ARM, "Security technology building a secure system using trustzone technology (white paper)," *ARM Limited*, 2009.

[29] W. Diehl, "Attack on AES Implementation Exploiting Publicly-visible Partial Result." *IACR Cryptology ePrint Archive*, vol. 2017, p. 788, 2017.

[30] K. Schramm, G. Leander, P. Felke, and C. Paar, "A Collision-Attack on AES: Combining Side Channel-and Differential-Attack," *Joye and Quisquater [8]*, pp. 163–175.

[31] C. Ashokkumar, B. Roy, M. B. S. Venkatesh, and B. L. Menezes, "" S-Box" Implementation of AES is NOT side-channel resistant." *IACR Cryptology ePrint Archive*, vol. 2018, p. 1002, 2018.

[32] H. Gamaarachchi and H. Ganegoda, "Power Analysis Based Side Channel Attack," *arXiv preprint arXiv:1801.00932*, 2018.

[33] Y. Huang and P. Mishra, "Trace buffer attack on the AES cipher," *Journal of Hardware and Systems Security*, vol. 1, no. 1, pp. 68–84, 2017.

[34] B. Ege, A. Das, S. Gosh, and I. Verbauwhede, "Differential scan attack on AES with X-tolerant and X-masked test response compactor," in *2012 15th Euromicro Conference on Digital System Design*. IEEE, 2012, pp. 545–552.

[35] T. Reece and W. H. Robinson, "Analysis of data-leak hardware Trojans in AES cryptographic circuits," in *2013 IEEE International Conference on Technologies for Homeland Security (HST)*. IEEE, 2013, pp. 467–472.

[36] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware Trojans in third-party digital IP cores," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, 2011, pp. 67–70.

[37] A. A. Pammu, K.-S. Chong, W.-G. Ho, and B.-H. Gwee, "Interceptive side channel attack on AES-128 wireless communications for IoT applications," in *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 2016, pp. 650–653.

[38] W. Hu, A. Ardeshiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property specific information flow analysis for hardware security verification," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[39] "What Constitutes a Breach of Confidentiality?" [Online]. Available: https://www.upcounsel.com/what-constitutes-a-breach-of-confidentiality

[40] A. Nahiyan, M. Sadi, R. Vittal, G. Contreras, D. Forte, and M. Tehranipoor, "Hardware trojan detection through information flow security verification," in *2017 IEEE International Test Conference (ITC)*, pp. 1–10.

[41] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *Journal of Hardware and Systems Security*, vol. 1, no. 1, pp. 85–102, 2017.

[42] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *2013 IEEE 31st international conference on computer design (ICCD)*. IEEE, 2013, pp. 471–474.

[43] "What makes hardware secure?" [Online]. Available: https://www.techdesignforums.com/practice/technique/are-you-formally-secure/

[44] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-based design*. Springer Science & Business Media, 2004.

[45] M. Boulé and Z. Zilic, "Automata-based assertion-checker synthesis of psl properties," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 1, p. 4, 2008.

[46] "Common Weakness Database for Hardware." [Online]. Available: https://cwe.mitre.org/data/definitions/1194.html

[47] O. Girard, *MSP430*, 2009 (accessed 04 Aug 2009). [Online]. Available: https://github.com/olgirard/openmsp430

[48] Z. Ning and F. Zhang, "Understanding the security of ARM debugging features," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 602–619.

[49] W. Chen and J. Bhadra, "Striking a balance between SoC security and debug requirements," in *2016 29th IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2016, pp. 368–373.

[50] H. Wang, H. Li, F. Rahman, M. M. Tehranipoor, and F. Farahmandi, "SoFI: Security Property-Driven Vulnerability Assessments of ICs Against Fault-Injection Attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[51] Cadence, *JasperGold SPV*. [Online]. Available: https://www.cadence.com/content/cadence-www/globl/en_US/home/tools/systemdesignand-verification/formalandstaticverification/jaspergoldverification-platform/securitypathverificationapp.html

[52] *Trust Hub Benchmarks*. [Online]. Available: https://trust-hub.org/home

[53] A. Vafaei, A. Hooten, M. Tehranipoor, and F. Farahmandi, "Symba: Symbolic Execution at C-level for Hardware Trojan Detection," *International Test Conference (ITC)*, 2021.

[54] B. Ahmed, F. Rahman, M. Tehranipoor, and F. Farahmandi, "AutoMap: Automated Mapping of Security Properties Between Different Levels of Abstraction in Design Flow," *International Test Conference (ITC)*, 2021.