Automated Assertion Checker Generator for Security Applications at the Register Transfer Level

Miguel Angel Alfaro Zapata

Department of Electrical & Computer Engineering

McGill University

Montréal, Québec, Canada

October 18, 2023

A thesis presented for the degree of Masters of Electrical Engineering

Abstract

In the midst of the growing complexity of designs and the increasing number of hardware security vulnerabilities, an efficient design verification strategy becomes essential. Initially, validation efforts focused on functional verification to ensure the design behaved according to the specifications. A fundamental limitation of traditional functional verification is its limitation to identify security vulnerabilities such as Information Leakage (IL) and Illegal States and Transitions (IST).

In this thesis, we present an automated tool that integrates Information Flow Tracking (IFT) techniques, Finite State Machine (FSM) detection, assertion generation, and hardware assertion checker generator to automatically produce security hardware checkers against IL and IST vulnerabilities on Register Transfer Level (RTL) designs. The tool is divided into two parts. The first augments the RTL models with IFT logic that models the information flow and automatically generates security assertions based on specified secure assets. The second one detects the FSMs present in RTL models and generates security assertions based on authorized and protected states. These assertions are then converted into hardware assertion checkers and merged into the baseline RTL model.

The tool was tested in various Verilog designs, including a microcontroller, two processors, a UART module, and an encryption core, thereby underlining its utility in various scenarios. We evaluate the impact of our tool on the slice logic, LUT, and FF quantity and compare them with the baseline design. We use the Vivado Design Suite to synthesize the RTL code and generate a resource utilization report. We report a maximum increase in slice logic utilization of up to 10.58% and 33.33% for explicit and implicit tagging, respectively. The integration of security assertion checkers produced a resource utilization overhead of 13.17% for the explicit tagged design and 11.52% for the implicit tagged design. The results demonstrate the successful generation of security assertions and assertion checkers with a medium impact on the utilization of FPGA resources while providing a systematic approach to security verification in hardware designs.

Abrégé

Face à la complexité croissante des conceptions et à l'augmentation du nombre de vulnérabilités en matière de sécurité matérielle, une stratégie efficace de vérification des conceptions devient essentielle. Dans un premier temps, les efforts de validation se sont concentrés sur la vérification fonctionnelle afin de s'assurer que la conception se comportait conformément aux spécifications. Une limitation fondamentale de la vérification fonctionnelle traditionnelle est son incapacité à identifier les failles de sécurité telles que les fuites d'informations (IL) et les états et transitions illégaux (IST).

Dans cette thèse, nous présentons un outil automatisé qui intègre des techniques de suivi du flux d'informations (IFT), la détection de machines à états finis (FSM), la génération d'assertions et le générateur de vérificateurs d'assertions matérielles pour produire automatiquement des vérificateurs matériels de sécurité contre les vulnérabilités IL et IST sur les conceptions de niveau de transfert de registre (RTL). L'outil est divisé en deux parties. La première complète les modèles RTL avec la logique IFT qui modélise le flux d'informations et génère automatiquement des assertions de sécurité basées sur les actifs sécurisés spécifiés. La seconde détecte les FSM présents dans les modèles RTL et génère des assertions de sécurité basées sur les états autorisés et protégés. Ces assertions sont ensuite converties en vérificateurs d'assertions matérielles et fusionnées dans le modèle RTL de base.

L'outil a été testé dans diverses conceptions Verilog, y compris un micro-contrôleur, deux processeurs, un module UART et un noyau de cryptage, soulignant ainsi son utilité dans divers scénarios. Nous évaluons l'impact de notre outil sur la logique de tranche, la LUT et la quantité de FF et les comparons à la conception de base. Nous utilisons Vivado Design Suite pour synthétiser le code RTL et générer un rapport d'utilisation des ressources. Nous signalons une augmentation maximale de l'utilisation de la logique de tranche allant jusqu'à 10,58% et 33,33% pour le marquage explicite et implicite, respectivement. L'intégration des vérificateurs d'assertions de sécurité a produit un surcoût d'utilisation des ressources de 13,17% pour la conception étiquetée explicite et de 11,52% pour la conception étiquetée implicite. Les résultats démontrent la réussite de la génération d'assertions de sécurité et de vérificateurs d'assertions avec un impact moyen sur l'utilisation des ressources FPGA tout en fournissant une approche systématique de la vérification de la sécurité dans les conceptions matérielles.

Acknowledgements

I am indebted to my thesis supervisor Zeljko Zilic, whose constant guidance and sustained patience have been the bedrock of this thesis. His scholarly wisdom and inspiring mentorship have propelled my academic evolution, making this achievement possible.

Regarding financial aid, I am grateful to Mitacs, NSARC, and Knox Inc. for generously sponsoring this research.

My sincere thanks are extended to the research lab team for a constructive environment pivotal to the realization of this thesis. In particular, I thank Tom for providing me with research ideas, constant feedback, and providing an intellectual environment through our lab meetings. Special thanks go to Hanneli and Katyayani for advising on the project, organizing our fruitful work sessions, and providing a sense of urgency through this Master.

Deep appreciation is due to Israel and Vivian, cherished friends from my Master's program. Together, we navigated challenging periods, helped each other during academic complexities, and provided constant support. Their contributions have enriched this journey in unique ways.

Finally, I offer unending gratitude to my parents and brother for teaching through actions, their unwavering dedication, constant emotional support, and signs of affection. They have modeled education through their actions and transformed our home into a haven of resilience. I share my triumphs with them, and we collectively learn from our setbacks.

Preface

This section declares that the work presented in this document was completed and carried out by Miguel Angel Alfaro Zapata. The author developed the implementation of the IFT and FSM detectors for this work. The tagging algorithm used in the IFT generator is derived from the work of Wei Hu and Armaiti Ardeshiricham, which is duly recognized. The author fully implemented the security assertion generator, adapted from the research papers of Hasini Witharana and Prabhat Mishra. The Hardware Assertion Generator (MBAC) used to synthesize the assertion checkers from System Verilog Assertions (SVA) was developed by Marc Boulé and Zeljko Zilic, a former Ph.D. scholar and the supervisor of the research group, respectively. The integration of open source tools, implementation of tagging techniques, automatic assertion generation, and hardware assertion merger were developed and debugged individually by myself.

Contents

List of Figures

List of Tables

List of Acronyms

ABV	Assertion-Based Verification.
AES	Advanced Encryption Standard.
ASIC	Application-Specific Integrated Circuit.
AST	Abstract Syntax Tree.
BSV	Bluespec System Verilog.
CAD	Computer-Aided Design.
CPU	Central Processing Unit.
DUV	Device Under Test.
EDA	Electronic Design Automation.
FF	Flip-Flop.
FOSS	Free Open-Source Software.
FPGA	Field Programmable Gate Array.
FSM	Finite State Machine.
GLIFT	Gate Level Information Flow Tracking.

HDL	Hardware Description Language.
IC	Integrated Circuit.
IFT	Information Flow Tracking.
IL	Information Leakage.
IoT	Internet of Things.
IP	Intellectual Property.
IST	Illegal States and Transitions.
LHS	Left-Hand Side.
LUT	Look Up Table.
N/A	Not Applicable.
PCI	Peripheral Component Interconnect.
PIC	Programmable Interrupt Controller.
PSL	Property Specification Language.
RTL	Register-Transfer Level.
RTLIFT	Register Transfer Level Information Flow Tracking.
SMV	Statistical Model Validation.
SoC	System-on-a-Chip.
SV	System Verilog.
SVA	System Verilog Assertions.
TLM	Transaction-Level Model.

UART	Universal Asynchronous Receiver-Transmitter.
VHDL	VHSIC Hardware Description Language.

Chapter 1

Introduction

1.1 Motivation

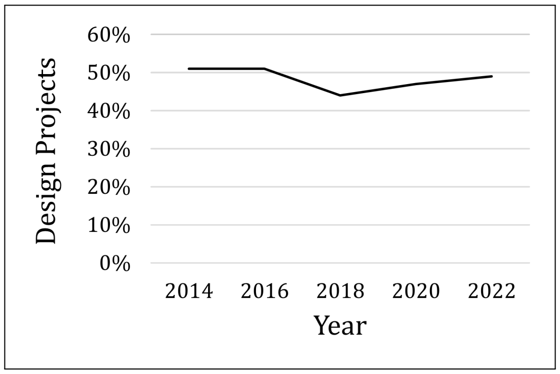
In an increasingly interconnected world, where computing devices permeate every aspect of our lives, the security of hardware components has become critical. Electronic designs must be verified to ensure that they comply with the requirements specified during the concept definition. For years, verification techniques have been instrumental in hardware design. However, as design complexity increases, field programmable gate array (FPGA) capacities expand, and time-to-market requirements shorten, traditional verification and validation techniques are becoming more resource-consuming and inadequate to achieve first silicon success (obtaining a post-manufactured integrated circuit (IC), application-specific integrated circuit (ASIC), or System-on-a-Chip (SoC) without bugs) [3].

A series of reports about the trends in functional verification of ICs, ASICs and FPGAs carried out since 2014 [4–8] showed that the proportion of time spent in validation activities has only increased (Fig. 1.1a). Despite the resources destined for verification, only around half of the projects reach the market within schedule (Fig. 1.1b). Moreover, the percentage of projects that reached first silicon success has not improved over the years (Fig. 1.1c).

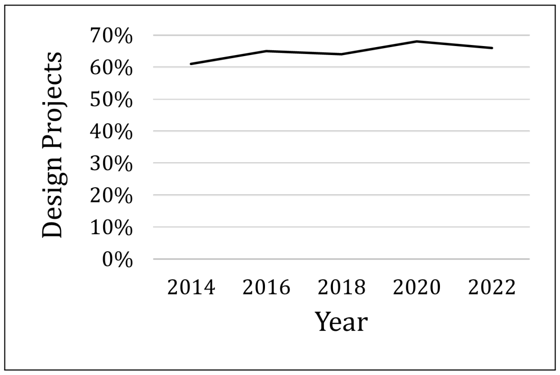
The latest study available in 2022 revealed that the design engineers spent 42% of their time in verification-related activities. From the time destined for verification, engineers spent 47% of their time debugging and 19% creating tests and running simulations [7]. However, despite the considerable allocation of resources and intensive verification efforts, only 30% of the FPGA projects were completed before or on schedule, and 17% of the projects achieved the first silicon pass [8].

Assertions are an integral part of the verification tools utilized during both simulation and formal verification. These assertions capture the behavior specified in the requirements and produce a signal error if the asserted conditions are violated. As manual assertion development can be time-consuming and error-prone, efforts have been implemented to automate assertion generation [9, 10].

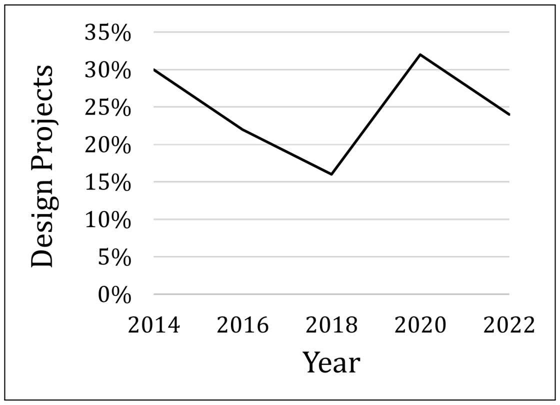
Despite the effort invested in creating and validating assertions, they are utilized mostly in the RTL stages and rarely used in future stages. An alternative explored for functional assertions is the conversion of the assertions to hardware assertion checkers that can be synthesized, simulated, and even integrated into the design.



(a) Time spent on verification.



(b) Projects behind planned schedule



(c) First silicon success rate.

Figure 1.1: Functional verification trends on IC and ASIC.

Initial efforts focused on functional verification to ensure that the hardware design behaves according to the specifications. However, functional verification is limited to the expected behavior of the design and might not detect hardware security vulnerabilities. For example, among the causes of bugs, 6% were associated with security features [8]. To identify these bugs and mitigate related vulnerabilities, security requirements have been integrated into the verification stages to address potential vulnerabilities that could be exploited by malicious actors [11].

This thesis seeks to integrate and expand the available tools used for hardware verification while enriching the open-source community. It is motivated by the need to augment traditional validation efforts to ensure the robustness of hardware design in terms of security. We aim to contribute to the creation of automated tools that aid the verification engineers and speed up the security verification of hardware designs.

1.2 Contribution To Knowledge

This thesis introduces an automated assertion checker generator that integrates Information Flow Techniques (IFT) techniques, Illegal States and Transitions (IST) detection, automated assertion generation, and hardware assertion checker generation to effectively model the information flow, detect Information Leakage (IL) and IST vulnerabilities. The objective of this research is to streamline the generation of security assertion checkers by implementing and integrating the techniques mentioned before. We contribute to the existing body of knowledge in the following ways:

- We present a detailed explanation of our IFT implementation for RTL designs.
- We integrate and adapt the tools and techniques related to RTL-IFT, FSM extraction, generation of security assertions, and hardware assertion checker generator into an automated tool.
- We evaluate our tool on five RTL designs: RS232 [12], AES-T1100 [12], CPU8080 [13],

PIC16F84 [12], and VexRISCV [14].

- We present experimental results to show the impact of our proposed tool on the utilization of FPGA resources.
- We let designers generate and integrate security assertion checkers into Verilog designs with minimal effort.

1.3 Document Structure

The structure of this thesis is composed of five chapters. Chapter 2 introduces the background knowledge and examines the literature on the strategies adopted in this work. Chapter 3 describes the implementation details of the Automated Assertion Checker Generator tool for both information leakage vulnerabilities and illegal states and transitions vulnerabilities. Chapter 4 reports the experimental results obtained for different embedded systems. The conclusions and future work are presented in Chapter 5.

Chapter 2

Background

This chapter presents the definitions and concepts necessary to understand the details of the investigation. The first section provides an overview of the hardware design cycle, security vulnerabilities, and examples of attacks that exploited these vulnerabilities. The second section explores verification techniques used in hardware designs. The third section reviews the available literature related to mitigation strategies for IL and IST, such as IFT, automated assertion generation, and hardware assertion checker generation.

2.1 Security Vulnerabilities

As shown in Fig. 2.1, the general hardware circuit design cycle begins by gathering and generating the requirements related to the application of the circuit and producing the specifications that capture the functional intent of the system. This set of specifications is transformed into High-Level Architectural (HLA) descriptions using Transaction Level Modeling (TLM), such as SystemC [15]. Subsequently, they are refined into an RTL implementation using Hardware Description Languages (HDL), such as Verilog [16]. The implementation is synthesized into a gate-level implementation and then into a layout through place and route tools. In the last stage, the design is converted into a layout to manufacture IC, ASIC, or SoC [17].

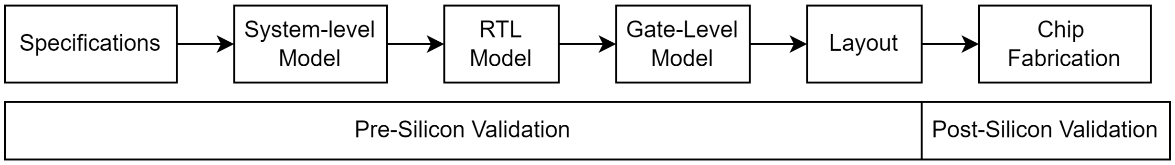


Figure 2.1: Hardware design cycle.

Every stage in the cycle can introduce errors or faults, either by the designer, by computer-aided design (CAD) tools, or by the usage of intellectual property (IP) created by third-party vendors, which can make the system vulnerable to attacks [18]

These faults may arise from practical or theoretical problems [19]. Practical problems refer to the context in which the design is performed, the decisions made by the designers, or the information available at the time of the specification stage. Technically, practical issues can be solved with sufficient time, knowledge, and resources. Examples of practical problems include design flaws, incorrect assumptions, or insufficient threat models. Theoretical problems originate from the models used to build the design and the design cycle (e.g., the abstraction process transforms a logical model into a physical system). For example, high-level design can miss features present at the physical level, so even if the information flow was verified at the system level, the physical implementation could leak out through physical aspects such as electromagnetic emission, power variation, or thermal activities [19].

Hardware vulnerabilities can be classified into six classes: malicious implants, IL, IST, permissions and privileges, resource management, and buffer issues.

- Malicious Implants: Similarly to code injection in the software domain, malicious implants, also called hardware Trojans, are hardware inserted by untrusted parties during the design or fabrication stages and designed to disrupt the system, making it vulnerable to other attacks. They are usually placed in areas of the SoC that are activated under specific circumstances, making them difficult to detect using standard validation techniques [20].
- Information leakage: Modern systems differentiate secure and non-secure information and features. If the information flow policies are not followed, secure information leaks to untrusted elements, or untrusted actors can gain access to private spaces in memory. For example, the XBOX gaming console transmitted a secret key through a bus within the chip, which attackers could access [21].
- Illegal States and Transitions: A secured SoC can be modeled as an FSM with protected states (which control privileged functions), authorized states (which are the only states

allowed to transition to a protected state), and unauthorized states. However, some FSM might include don't care or undefined states, leading to unexpected behavior of the system. These vulnerabilities can be exploited through fault injection attacks or malicious implants [22]. The work by Gaubatz et al. highlights the importance of developing mitigation strategies against these attacks for cryptographic applications [23].

- Permissions and privileges: Permissions and privileges are critical components of access control subsystems, where resource control is restricted to specific permission levels, for example, user mode, interrupt mode, and supervisor mode. Taking advantage of the IST vulnerability, the FSM controlling the rules to access a protected state could be bypassed, causing privilege escalation attacks [24].

- Resources management: Certain resources, such as the debugging infrastructure, have special access to memory and hardware. There have been cases where these resources are left enabled when distributed to the market, which allowed attackers to take over the device. For example, it was found that the FPGA Microsemi ProASIC3 had a vulnerability on the JTAG chip, which led it to enable the debugging infrastructure and bypass the chip's security [25].

- Buffer Issues: Buffers are generally used in SoC designs for communication and features such as out-of-order execution [18]. These buffers might expose information if not flushed after branch prediction or if the content can be modified, affecting the processor's behavior.

This research focuses on IL vulnerabilities to ensure the privacy of sensitive data.

2.1.1 Information Leakage

Information security is described as the protection of information and information systems against unauthorized access, usage, disclosure, disruption, alteration, or destruction [26]. It consists of three aspects: confidentiality, integrity, and availability. Confidentiality protects the information and ensures that it remains accessible solely to authorized entities. Integrity involves protecting against unauthorized alterations or destruction of information. It includes both the content of the data and the source of it. Lastly, availability refers to the timely and reliable access and utilization of information and resources [19].

Information leakage, or data leakage, is a direct attack on confidentiality. In the context of hardware vulnerabilities, it refers to the unauthorized transfer or disclosure of sensitive information, such as cryptographic keys, passwords, or classified data, which provides a stepping stone for further attacks on the system. As surveyed by [19], different factors cause IL vulnerabilities. For example:

- The increasing complexity and details involved in the design of modern SoCs. As the size and components of the system increase, the errors and verification time grow, making it harder to detect and solve any issues.

- Inappropriate security policies for users and data objects. For example, giving administrator privileges to users, not updating the system, or storing information in non-secure spaces.
- Process of abstraction: A higher-level model does not consider properties of interest at a lower level. For example, side-channel attacks include acoustic, electromagnetic emission, power variation, or thermal activities.
- Covert channels: In the context of program confinement, a covert channel happens when information is exchanged between two parties that are neither allowed to communicate nor intended to transfer data. They can be classified into two types depending on the nature of the covert channel.
  - Covert timing channels: They exploit the temporal characteristics of events to transfer information. By observing the time taken by the CPU, attackers can extract information about the values of bits or registers.
  - Covert storage channels: These are channels that allow the transmission of information between storage objects of different security classes, which should be otherwise segregated. This is done using mechanisms that were not originally designed to function as information channels.

2.1.2 Illegal States and Transitions

The behavior of an SoC can be described using FSMs, where a control signal or a set of control signals manage the functionality of the design [22]. In the case of a secured SoC, the FSM includes protected, authorized, and unauthorized states. As mentioned, protected states handle private information or perform privileged operations. It is worth noting that any other state, unauthorized or undefined, is considered an illegal state [2]. Protected states should only be reached by authorized states and be inaccessible to illegal states. For example, Fig. 2.2 shows a simple FSM with four states. If we assume that state 3 is protected, state 2 is authorized, state 3 is unauthorized, and state 4 is undefined. Only state 2 should transition to state 3, while states 1 and 4 would be considered illegal.

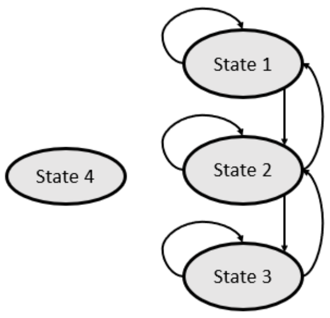


Figure 2.2: FSM example with four states.

While functional validation covers the expected FSM's behavior (valid states), it might miss illegal states and transitions vulnerabilities produced by flawed error-handling mechanisms, logic flaws, input validation weaknesses, fault injection attacks or malicious implants. This enables attackers to access protected states and compromise the system [2].

2.2 Verification Techniques

Verification techniques are used to ensure that the design and implementation of hardware systems are correct and to detect the presence of bugs and potential hardware vulnerabilities. These techniques can be classified into formal and dynamic.

Formal techniques use mathematical methods to analyze the hardware design exhaustively and to prove the correctness of a design. There are two main approaches in formal techniques: equivalence and model checking. Equivalence checking proves that a reference design is logically equivalent to an alternate representation. It transforms designs into mathematical models and compares registers, inputs, and outputs to prove that both representations exhibit the same behavior [27]. The model checking uses a logical model (functional specification describing the relation between inputs, outputs, and internal states) of the circuits to check if it satisfies a set of formal properties [28].

However, formal techniques come with their limitations. One of them is the state-explosion problem. The number of global states in a system with multiple processes can expand exponentially as the size of the design increases, which hampers the practical usefulness of these techniques. If the number of states is too large, the time and resources required to complete the verification procedure make them unusable [29]. Another obstacle is the overhead of adopting these techniques. For example, model checking requires specifying an initial set of axioms, creating a set of inference rules derived from these hypotheses, and creating the logical model [29].

On the other hand, dynamic techniques involve executing the design using a large number of input stimuli (test vectors) and observing the behavior of the design under verification (DUV) [30]. Some dynamic techniques include simulation-based, code coverage, functional coverage, FPGA-based prototyping, and Assertion-based Verification (ABV).

In the case of simulation-based verification, the capacity to catch functional errors relies on the quality and quantity of the test vectors. Since overall time-to-market requirements restrict the time spent testing a design, some bugs might remain in the design if they are in hard-to-activate scenarios [18]. The initial strategies used random generation, where a subset of all possible values of the test vector is chosen. This technique is fast and scalable, but does not ensure the activation of specific scenarios [18]. If the verification team knows the specific scenarios that should be tested, they can develop tailored test vectors, also called direct testing. However, manual development can be time consuming, relies on the designer's expertise, and is prone to errors [31]. To overcome these challenges, new techniques have been developed to improve test vector generation, such as constrained randomization testing, concolic testing, and machine learning.

Emulation-based verification uses an emulation platform (emulator) and specialized hardware accelerators to model the behavior of a digital design. The model works at a lower level of abstraction than simulations. The RTL design is mapped onto the emulator, which can execute the RTL design at a much higher speed than conventional simulation tools. The main disadvantage of an emulator is the higher cost of the equipment needed, which restricts the availability for small companies and researchers in academia [32].

In FPGA-based prototyping, the RTL model is executed on an FPGA in real-time while giving bit-accurate information about the DUV. The RTL model is transformed into a gate netlist and then into a bitstream used to program an FPGA device. FPGA-based prototyping offers different advantages over simulation and emulation. The execution speed is faster, allowing the use of interfaces (e.g. USB, DDR, PCI, etc.), and it has a minor cost compared to emulation-based verification. However, this approach offers poor visibility into the design and requires hardware and software expertise to implement the RTL on FPGAs [32].

ABV uses assertions to complement simulation-based verification and formal verification. It embeds these assertions into the design code or the verification environment to check the state and relationships of different signals. These assertions monitor the actual behavior of the design, and indicate if a violation occurs, which suggests the presence of bugs or design flaws [33]. A more detailed explanation of the assertions is provided below.

2.2.1 Assertions

Assertions are statements that specify the correct behavior of a design. It uses properties specified by temporal logic and a generalized form of regular expressions to provide an unambiguous description of the specifications. When interpreted by verification tools, they can be interpreted as executable specifications. Assertions should be created at the specification stage to formally document the system requirements that can be used in the other stages of the design lifecycle. They are typically used in dynamic verification to monitor the signals in real-time, which eases the checking and debugging process. Assertions are also used in model check verification to describe formal properties [34].

Assertions can represent a complex range of behaviors using Boolean expressions, extended regular expressions, and a large set of temporal operators [35]. The two main assertion languages used in hardware verification are SVA [36] and PSL [37]. Assertions languages have a complex set of syntax and semantics that are beyond the scope of this thesis. However, we describe the components and characteristics of the assertions to understand the assertions generated in this work. We will focus on SVA in the rest of the section since it is the language used for this research. A detailed explanation of each operator can be found in [34].

An SVA statement, such as the one shown in Fig. 2.3, is composed of the following parts:

- Boolean layer: This is the most basic layer of an assertion. It represents Boolean expressions with simple conditions or relationships between signals or variables. Boolean expressions represent true/false conditions. They include identifiers, unsigned numbers, Verilog parameters, and standard arithmetic, logic, and bitwise operators.

- Temporal layer: Expresses behaviors that can span over time.

  – Sequences: They describe a series of events over time in relation to other expressions. These are used to form temporal chains of events of Boolean expressions. Some of the SVA operators used for sequences are consecutive repetition ($*$), non-consecutive repetition ($=$), Goto repetition ($- >$), delays (##), throughout and, or, within and intersect.

  - Properties: Properties group Boolean expressions and sequences to form complex temporal behavior. Some of the property operators are: not, or, and, if/else, overlapped suffix implication ($|- >$), and nonoverlapped suffix implication ($| =>$).

- Disable clause: It is an optional directive that specifies the event that can turn off the assertion check. For example, during the reset of a system, when the results are not valid or not meant to be intended to be checked.

- Assertion clock: It specifies the name and edge of the clock for sampling values of the property where the assertions are evaluated.

- Verification layer: They provide the commands that indicate to the verification tool that the properties should be checked against the running hardware and report any failure. The verification directives are assert, covert, assume, or restrict. The assert property indicates that the property must hold, while the cover property indicates that the property must complete successfully at least once during the verification process.

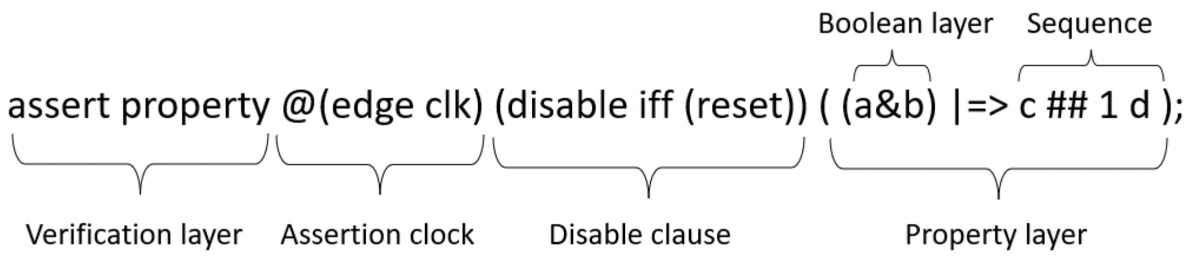


Figure 2.3: SVA examples and components.

2.3 Literature Review

2.3.1 Information Flow Tracking

As mentioned in Section 2.1.1, information can leak through covert storage channels, where objects that do not usually store data but control the information flow can leak information between processes. A widely used tool to detect IL is IFT, which models data propagation through a system and verifies that only authorized information flows occur [38]. IFT models the information flow using a tag-and-taint technique. First, the original design is tagged using a set of labels that store the security class of each signal. The user then assigns a value to each label, tainting them according to their security policies. During verification, the propagation of each label is monitored to ensure that the security policies are followed [39]. If at the end of the simulation, an non-secure output receives a tainted value (e.g., private information), we can assume that the system is leaking information.

Flow Model

IFT can be modeled using a flow model $\mathcal{FM}$ using the definition by Denning [1], summarized in Fig. 2.4.

```
𝓕𝓜 = <𝓝,𝓟,𝓢𝓒,⊕,→>
𝓝 = Storage Objects
𝓟 = Processes where information flows
𝓢𝓒 = Set of security classes
⊕ = Class combining operator
→ = Allowed information flows
```

Figure 2.4: Flow model elements.

A storage object ($\mathcal{N}$) is any element that can take or store values. Depending on the level of abstraction, they can be files, inputs, outputs, wires, registers, signals, flip-flops, memory blocks, etc.

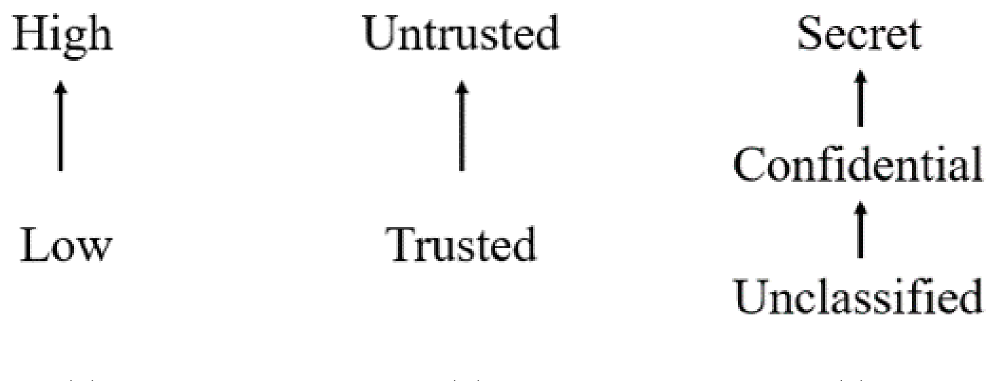
A process ($\mathcal{P}$) is any component that operates on these objects and produces a resulting object. Examples of processes are arithmetic, logic, and assignment operations.

A security class ($\mathcal{SC}$) corresponds to the security classification of data objects defined by the information flow policy. The $\mathcal{SC}$ are stored in tags or labels linked to each signal. For example, if signal ”a” is considered secure and signal ”b” is considered non-secure, we can define ”a_t” as the security class object of signal ”a” with a value of 1 and ”b_t” as the security class object of signal ”b” with a value of 0.

The class-combining operator ($\bigoplus$) calculates the output security class after combining two or more security classes. In security information applications, the output $\mathcal{SC}$ is given by the disjunction operation of the $\mathcal{SC}$ of the inputs. For example, suppose that we have the following process a = b + c. The security class of "a" (a_t) is given by b_t | c_t, and if at least one of the inputs belongs to the "secure" $\mathcal{SC}$, then a_t is "secure" as well, meaning that secure information is flowing to signal a.

Finally, the flow relation operator ($\rightarrow$) defines which information flows are allowed and which are prohibited, and is given by the information flow policy. For example, "A $\rightarrow$ B" indicates that the flow of information from A to B is permitted, while "B $\nrightarrow$ A" prohibits the flow from B to A.

The information flow policy can be modeled by the lattice of the system. A lattice can be given in the form of $\mathcal{L} = \{\mathcal{E}, \sqsubseteq\}$. $\mathcal{E}$ is the set of elements of the lattice, and $\sqsubseteq$ is a partial order relation of $\mathcal{E}$. The lattice must specify at least two different elements, one being a least upper bound element and another one a greatest lower bound element. Examples of lattice structures used in security applications are shown in Fig. 2.5.



(a) (b) (c)

Figure 2.5: Examples of security lattice structures [1].

IFT Classification

IFT can be classified according to the information flow type (explicit or implicit) [39] and the operator precision (imprecise or precise).

An explicit flow takes into account information that is directly moved from a source operand to a destination operand (e.g. an assignment operation a = b). In contrast, an implicit flow also takes into account the information flow generated by context-dependent execution and conditional operations (if and case statements). In an implicit flow, information can be passed down between objects even when there is no explicit data assignment [39].

Consider the multiplexer in Fig. 2.6a. According to the explicit flow, the output $\mathcal{SC}$ depends on which signal is selected (either a_t or b_t), shown in Fig. 2.6b. However, if we consider the underlying structure of the multiplexer (Fig. 2.6c), we can notice that the signal ”sel” interacts with the signals and can provide information about the system. Therefore, in the implicit tagging, the output $\mathcal{SC}$ will depend on the $\mathcal{SC}$ in ”sel_t”, as shown in Fig. 2.6d.

The precision of the class-combining operator($\bigoplus$) refers to how the $\mathcal{SC}$ of the inputs combined and propagate to the output objects. An imprecise operator considers that if any $\mathcal{SC}$ is tainted, the output $\mathcal{SC}$ will be tainted. The imprecise operator might produce more false positives since it will propagate a tainted value even if there is no actual information flow.

A precise operator considers the context of the operation and the value of the input and

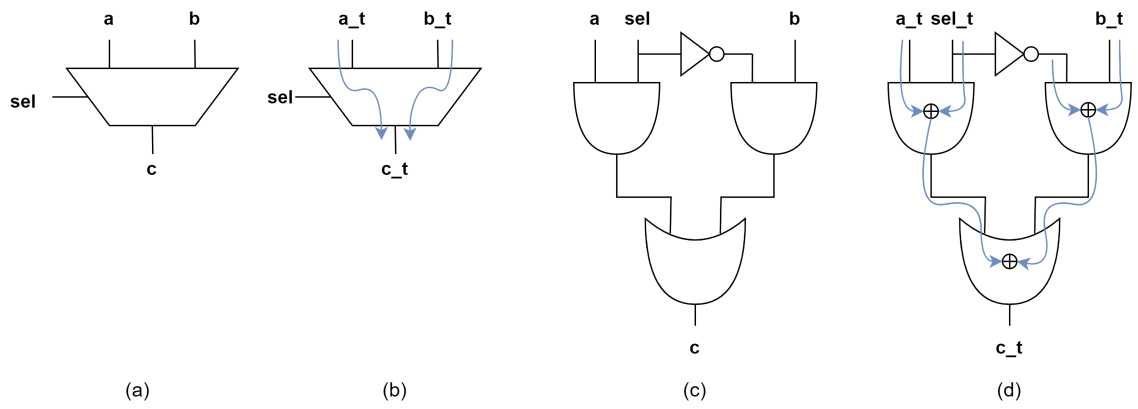


Figure 2.6: Multiplexer structure and Information flow types.

tagged signals. The precise operator takes into account the operation that takes place in the signals to calculate the output $\mathcal{SC}$. It is more accurate, but leads to more significant overhead to calculate the output $\mathcal{SC}$.

IFT Implementations

IFT was first implemented in software (operating systems [40], programming languages [41], cloud computing [42], etc.) and later adopted for hardware verification, often called Hardware IFT. The implementations of hardware IFT can be categorized according to the abstraction level used: software-mediated IFT, language extension, formal languages, gate-level IFT, cell-level IFT, or RTL-level IFT.

Software-mediated IFT monitors and enforces information security policies during runtime. Each instruction of the instruction set architecture (ISA) is modified (e.g., integrated pervasive processor modification or modular core additions or coprocessor support) to enforce the security policies specified by the user. Therefore, the information flow is monitored within the chip itself [38]. Two examples of software-mediated IFT are LIFT [43] and RIFLE [44].

At the language level, it can be implemented by creating a new type-enforced Hardware Description Language (HDL). These languages directly generate circuits that ensure the desired IFT properties by adding a typing system to an FSM. Subsequently, the designer has to assign a security label to each register [38]. They can use static types such as Caisson [45] or dynamic types to perform replication to restrict flow information, such as Sapper [46]. The main disadvantage is that designers must learn a new language and adapt their tools to the language, and any previous designs must be replicated in the new language.

Another approach is the HDL extensions, which expand the language capabilities to include security-related features, such as the ability to specify and enforce security policies, such as information flow control. However, the user still needs to specify the security label for each variable [38]. An example is SecVerilog [47], which extends the Verilog language and allows the user to implement the information tracking flow.

Formal languages can be used to represent HDL models. It eliminates the need to redesign the hardware but requires the user to annotate the resulting code to analyze the properties. An example is VeriCoq-IFT, which transforms Verilog designs into representations in the Coq language. [48]

At the gate level, each primitive is synthesized with additional IFT logic [49]. Gate-Level IFT (GLIFT) [50] works at the gate level after the design is translated into a netlist of lower-level components. A shadow gate is created for each of the gates synthesized from RTL, which tracks the flow of the tainted value of the original signals. Since the number of different gates is low compared to high-level structures, it is easier to generalize the tool. One of the limitations of GLIFT is that the tool does not scale well with the complexity of the design. Additionally, many high-level relationships get lost during synthesis, which limits the result analysis and the use of assertions.

In between the RTL level and the gate netlist, some synthesis tools create intermediary structures, e.g., Yosys creates an RTL Intermediate Language (RTLIL) [51]. The RTL model is transformed into this structure described by generic cells. CellIFT [52] uses the same principle as GLIFT, but at the RTLIL level, so it benefits of working at a higher level but still using general and repeatable structures.

RTLIFT [53] enables verification of security properties by performing a static analysis of the source code. RTLIFT receives Verilog files and the desired IFT precision level (explicit, imprecise, or precise), generating an equivalent Verilog code with the IFT logic added.

2.3.2 Assertion Generation

As mentioned in Chapter 2.2.1, assertions are a powerful and flexible tool for validating hardware designs. However, the manual generation of assertions tends to be time consuming and error prone. To streamline this process, different automated assertion tools have been developed. These techniques can be grouped into two approaches: static analysis of the source code or dynamic analysis of the simulation traces [2].

IODINE [54] automatically extracts properties from design simulations by detecting dynamic invariance for hardware designs. An invariant is a property that holds at a certain point in a simulation. These invariants might help to understand the program even if they were not considered in the property specification. The main drawback of invariants is that they might produce false properties since they are based on the testing vectors and the design being verified.

Dianosis [55] generates complex properties inferred from simulation traces and previously validated properties. The property generation is divided into two parts. First, they have a predefined set of basic properties that are inferred over the signals of the design. The design is simulated and the simulation trace is analyzed. If the generic properties are held, they are used for the second phase. The next part analyzes the temporal dependencies between properties. If the dependency is valid, it generates a more complex property that is recorded in a database. This process is repeated with new dependencies until no new properties are found.

SocVer [56] generates assertions based on a block-level structural analysis of the design. First, the HDL is parsed and the components of the system are divided into blocks. The information of each block is extracted (signals, ports, net names, attributes), and the blocks are classified into classes. Afterward, a predefined verification assertion schema or template is instantiated for each block, depending on its class. These assertion templates were previously specified in a library, but new schemas can be created.

Goldmine [9, 57] is an automatic assertion generator that combines static and data mining of simulation traces. First, the design is simulated using random vector generation. The data contained in the simulation trace are analyzed by A-miner [58], which combines a decision tree-based supervised learning algorithm and user-defined assertion templates to generate candidate assertions. Afterward, the assertions are evaluated using Statistical Model Validation (SMV) and ranked according to assertion coverage. One limitation of Goldmine is that it cannot guarantee that the generation assertions follow the design specification since they are derived from simulation traces and not a golden reference model.

The work in [59] proposes a five-step methodology for simulation-based verification to verify the control system of a universal asynchronous receiver-transmitter (UART) transmitter implemented as an FSM. First, they gather all interface signals (registers, inputs, and outputs). Then they list the functional spots to be verified, such as control blocks and FSMs, and the verification required for each spot. Then, using SVA, they describe the functions to be verified. Finally, they define properties to measure the functional coverage of the assertions.

In [60], Turumella et al. described an assertion-based verification approach that combines formal verification and simulation-based verification techniques. They tested this approach on a SPARC microprocessor. Although not fully automated, they proposed a methodology to use assertions derived from the design specification using OpenVera Assertions (OVA) [61].

Initially, efforts to automate assertion generation focused on functional verification of the system through ABV. As the occurrence of hardware attacks increased, so did the need to address vulnerabilities and security verification. Recent methodologies for the generation of security properties have leveraged the work done for functional validation.

The work of Witharana et al. [2], published in 2023, presents a vulnerability analysis for RTL designs along with an automated generator for security assertions. It offers a guide to address the primary security concerns in System-on-a-Chip (SoC), mainly discussed in section 2.1. They developed different algorithms for each vulnerability to generate assertions that detect them. Their framework can simplify security validation compared to manually generating security assertions. In the case of IL, they ensure that no design assets leak to non-secure outputs by performing a tag-and-taint analysis of the RTL code. Their algorithm consists of two steps. First, they tag the design using IFT, which will be described in detail in the next section. After the design is tagged and new variables are created, they produce the properties that will be asserted based on a list of secure assets specified by the designer.

2.3.3 Hardware Assertion Checker Generation

Since assertions written in SVA or PSL are not synthesizable, the knowledge gathered during the specification and design stages tends to be used only at the pre-silicon verification stages. However, assertions can still play an important role in the post-silicon stages. For example, due to resources (e.g. memory and computing power) and time limitations, pre-silicon verification might be insufficient to capture all possible use cases. Also, the behavior of higher-level models might defer from the actual silicon behavior due to asynchronous interfaces, potential manufacturing errors, timing errors, etc. Finally, security properties should be monitored during runtime to ensure enforcement at all times [62].

An alternative to re-use the assertions generated in previous stages is to create equivalent assertion checkers, also called online monitors. These checkers can be synthesized in the post-silicon stage to detect potential manufacturing errors during runtime. Additionally, previous validation stages may not be able to capture all scenarios [63]. Initial efforts focused on the manual creation of assertion checkers. However, since PSL and SVA are powerful languages, a single line of PSL or SVA can produce hundreds of lines of Verilog code [34]. Automatic generation of checkers is much more advantageous than designing checkers by hand. Different tools have been implemented to generate synthesizable Verilog code from assertions.

The work in [64] transforms assertions written in ANSI-C into synthesizable assertion checkers. Their approach converts assertion statements into an if-statement structure and an assertion notification function. When the assertion fails, an error signal is sent to the notification function. Their approach is unclear about which type of properties are supported and what the resource utilization overhead is. Additionally, their implementation is limited to the use of ANSI-C language to generate assertions.

A similar approach, proposed in [65], introduces a method to generate hardware modules in Bluespec System Verilog (BSV) from SVA. Their method translates each layer of the assertion into Bluespec commands, with the exception of the temporal logic, which is mapped into FSMs. Then, the baseline design and assertions, written in Bluespec System Verilog, are synthesized into a netlist with an increase of 9% circuit area. This technique requires the design to be written in BSV, which might not be compatible with other verification tools.

Finally, a widely documented and used tool is MBAC [34], an automata-based assertion checker generator. It produces resource-efficient and behaviorally correct assertion checkers. It is compatible with SVA and PSL assertions using linear temporal logic and produces synthesizable Verilog code optimized for a low resource utilization overhead.

2.3.4 Automated Security Assertion Checker Generation

After reviewing the available literature, we found a few research papers that integrate assertion checkers focused on security vulnerabilities. Table 2.1 summarizes the most relevant articles related to security verification using assertion checkers. The table includes articles that present IFT methodologies, assertion generation for functional and security applications, specifically assertions for information leakage and illegal states and transitions, and assertion checker generation. Despite the numerous publications about each of the topics, there are few attempts to streamline the generation of hardware assertion checkers based on security vulnerabilities, in particular for IL and IST.

Table 2.1: Articles related to automated security assertion checkers generation

Article Name	Information Flow Tracking	Assertion	Information Leakage	Illegal States Transitions	Assertion Checker Generation
Automated Generation of Security Assertions for RTL Models [2]	Yes	Security	Yes	Yes	No
Property Specific Information Flow Analysis for Hardware Security Verification [66]	Yes	Security	Yes	No	No
An Infrastructure for Debug Using Clusters of Assertion-Checkers [3]	No	N/A	No	No	Yes, for functional verification
Reusing Verification Assertions as Security Checkers for Hardware Trojan Detection [62]	No	Security	No	No	Yes, for security verification
Hardware Information Flow Tracking [39]	Yes	Security	Yes	No	No
Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design [38]	Yes	N/A	No	No	No
CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL [52]	Yes	N/A	No	No	No
Complete Information Flow Tracking from the Gates Up [50]	Yes	N/A	No	No	No
A Hardware-based Technique for Efficient Implicit Information Flow Tracking [67]	Yes	N/A	No	No	No
Automatic Generation of Complex Properties for Hardware Designs [55]	No	Functional	No	No	No
Block-based Schema-driven Assertion Generation for Functional Verification [56]	No	Functional	No	No	No
Automatic Generation of Assertions from System Level Design Using Data Mining [68]	No	Functional	No	No	No

Chapter 3

Proposed Methodology

This chapter will be organized as described below to examine how our proposed tool generates assertion checkers for information leakage.

- **Implementation Overview:** The following section provides an overview of the inputs and outputs of the tool and its information flow. It describes the relationship between the main components of the systems: the IFT generator, the FSM extractor, the assertion generator, and the assertion checker generator.

- **IFT Generator**: This section presents the information flow tagging methodology, adapted from the IFT model proposed in [38, 39]. First, it describes PyVerilog, a tool to parse Verilog code and generate an Abstract Syntax Tree (AST) [69]. We then describe the tagging algorithm used for the two different flow models: explicit and implicit. Lastly, we introduce the assertion generation algorithm used to generate the security assertions for IL.

- **IST Generator**: In this section, we describe each of the components of the automated assertion checker generator for IST. We present Yosys [70], a framework used to extract the FSM, the submodules used to adapt it to the Python environment, and the algorithm used for the automated assertion generation. Finally, we describe the algorithm used to create the security assertions for IST.

- **Automated Assertion Checker Generator:** Finally, we describe the submodule that integrates MBAC to Python and adapts the assertion file and the input design to MBAC's requirements [34]. This submodule is used by both the IFT and IST generators to produce a Verilog file with the assertion checkers merged into the design.

3.1 Implementation Overview

The automated assertion checker generators proposed in this work streamline the integration of assertion checkers into RTL designs with minimum user interaction. Additionally, it generates intermediate files that can be used independently for subsequent verification activities. Its core is built around Python, but different submodules and open-source tools are written in other languages. For instance, MBAC is written in C, Yosys is written in C++ and executed using command scripts, and the FSM information is extracted using regular expressions.

A simplified view of the automated assertion checker generator for information leakage is shown in Fig. 3.1. The gray boxes correspond to the three main sub-modules. The first submodule receives the source code written in Verilog and the tracking level flag. This flag specifies the type of information flow to be monitored (explicit or implicit). The IFT generator module tags the baseline design and creates a signal list with the information of the created tags. The designer should complement this list with information about the security assets and non-secure outputs that should be tracked.

The second submodule of the tool reads the list with the security asset information and generates the security assertions focused on IL vulnerabilities and saves them into a file. Our approach is flexible enough to allow for further inclusion of user-made assertions if needed. Then, the assertion checker generator converts the assertions in the file into hardware assertion checkers written in Verilog and integrates them into the tagged design.

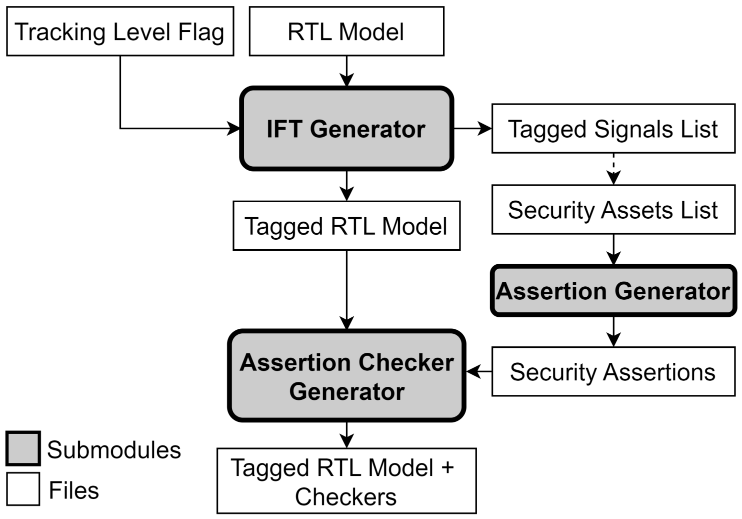


Figure 3.1: Overview of the automated assertion checker generator for IL.

A similar approach was followed for the automated assertion checker generator for IST, shown in Fig. 3.2. The assertion generator for IST reads the RTL model and the state specifications provided by the user. The state specifications describe which states are illegal, authorized, or secured. The submodule extracts the FSMs from the Verilog designs using the Yosys tool [70]. Based on the FSMs and the state specifications, the submodule creates a file containing the security assertions. With this information, the assertion generator creates a file containing security vulnerabilities that monitor the FSM states and transitions. Finally, the assertion checkers are generated and merged into the original RTL model inside the assertion checker generator submodule.

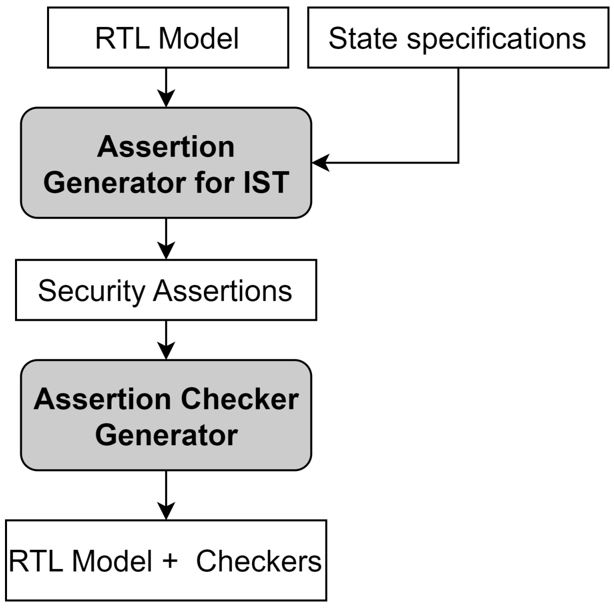


Figure 3.2: Overview of the automated assertion checker generator for IST.

3.2 IFT Generator

The IFT generator, shown in Fig. 3.5, comprises four main components: Verilog parser, clock extractor, tagging algorithm, and code generator. The Verilog parser and code generator are functions provided by PyVerilog [69]. The clock extractor and tagging algorithm were implemented with the help of utility tools and PyVerilog AST object templates. Since the rest of this section relies on the specific functioning of PyVerilog and some of its data structures, we describe the toolkit before going into the details of the IFT generator.

3.2.1 PyVerilog

PyVerilog is a Python-based open-source toolkit that facilitates the hardware design processing of Verilog-based projects. It includes a Verilog parser, a data flow analyzer, a control flow analyzer, and a code generator [69]. It has been used for RTL analysis and Hardware Security research such as [2, 71]. It is composed of five different libraries:

- Vparser: Verilog code parser and AST generator.
- Data flow Analyzer: AST analyzer to extract the signal information and generate a dataflow graph.
- Control flow: FSM detector and extractor.
- AST code generator: It contains the functions that create a file with a synthesizable Verilog design.

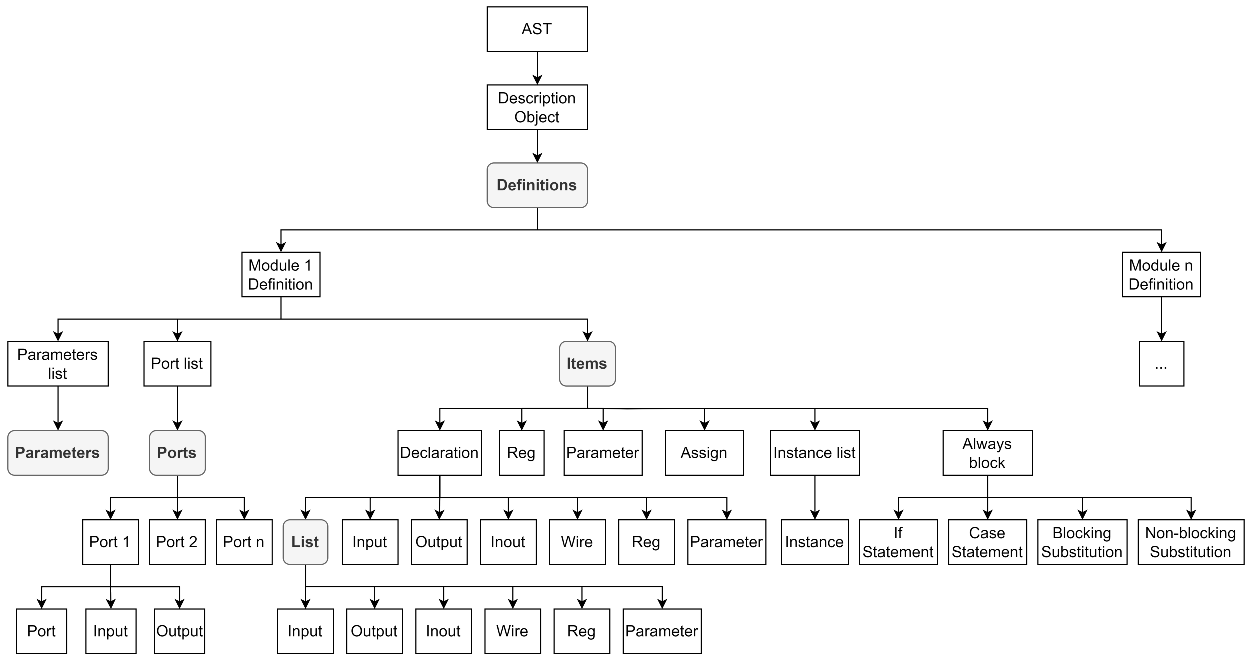
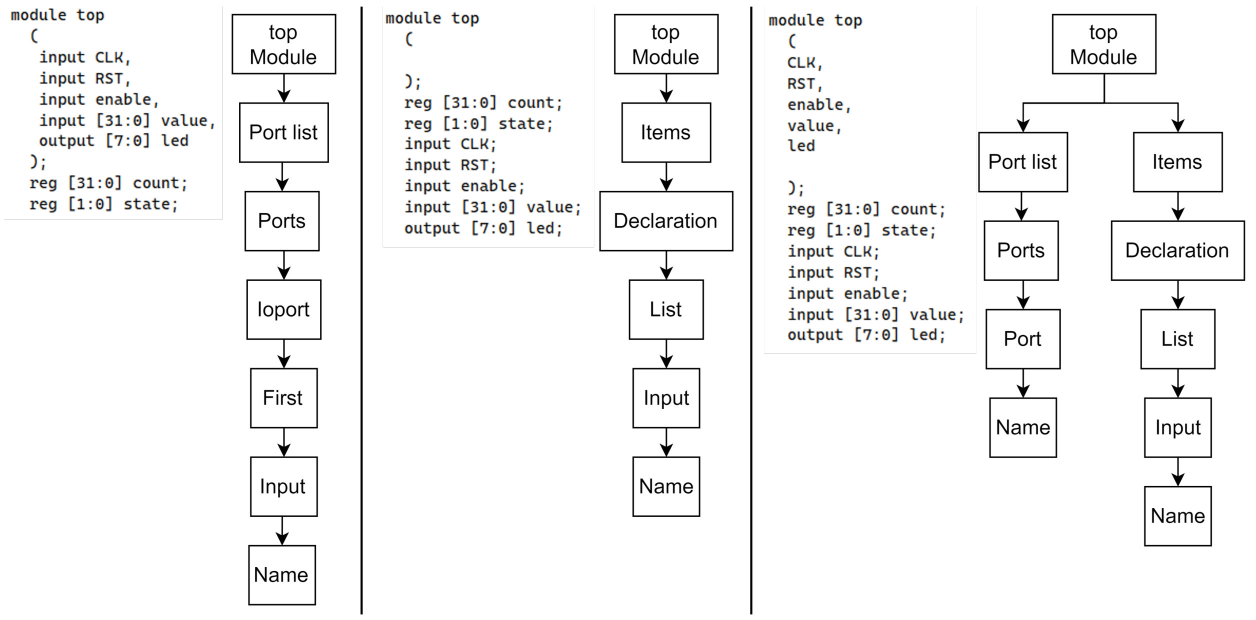


Figure 3.3: AST structure generated by PyVerilog.

- Utils: Miscellaneous utility tools for extracting information from the AST.

First, PyVerilog reads and converts one or more files containing the Verilog design into its equivalent abstract syntax tree (AST). The generated AST is stored as a Python object containing a module definition for each module in the RTL design. Each module definition object comprises three sections: Parameter list, port list, and item list. The content of each list is summarized in Fig. 3.3.

Due to the flexibility of Verilog, ports and signals can be declared in three different ways, which affect how they are stored in the AST: inside the port list of the module, as individual items, or as a list of ports. Fig. 3.4 shows the changes in the AST depending on how the ports are declared. A similar situation occurs for module instantiations, concatenations, the



```
module top
  (
   input CLK,
   input RST,
   input enable,
   input [31:0] value,
   output [7:0] led
  );
  reg [31:0] count;
  reg [1:0] state;
```

```
module top
  (

  );
  reg [31:0] count;
  reg [1:0] state;
  input CLK;
  input RST;
  input enable;
  input [31:0] value;
  output [7:0] led;
```

```
module top
  (
  CLK,
  RST,
  enable,
  value,
  led

  );
  reg [31:0] count;
  reg [1:0] state;
  input CLK;
  input RST;
  input enable;
  input [31:0] value;
  output [7:0] led;
```

Figure 3.4: Verilog code influence in AST structure for ports declaration.

use of parentheses, conditional statements, and always blocks. To explore and modify the AST, the user needs to be aware of this scenario and understand the AST structure to change it. The AST can be modified by removing, replacing, or adding items to their corresponding list. New items can be created by filling in the information from the object templates from the vparser library.

The data flow analyzer uses the AST generated by the parser to construct a data flow graph. The data flow analyzer comprises three subsystems: module analyzer, signal analyzer, and bind analyzer. First, the analyzer traverses the AST and lists all the modules. Then, it generates a list of signals and their information, such as signal type, size, declarations, and constant value definitions (when using parameters). Finally, the bind analyzer generates a data flow graph for each signal. The control flow analyzer later uses this data flow graph to create a graph representation of the FSM. The control flow analyzer uses a state machine pattern matching tool to identify signals linked to FSMs. Then, the active condition analyzer explores the signal's data flow graph and extracts the assignments and assignment conditions to generate the FSM.

The user can find a description of the main libraries, functions, examples, and an installation guide in the PyVerilog document [69] and in the GitHub repository [72].

First, the IFT Generator submodule parses the source code using the Pyverilog vparser library. Then, it generates an Abstract Syntax Tree (AST), whose structure is described in Fig. 3.3. The AST is analyzed using the data flow analyzer, which identifies the signals corresponding to a clock. This information is linked to the module and the signals. The tool extracts the system's clock and stores it in the tagged signals file.

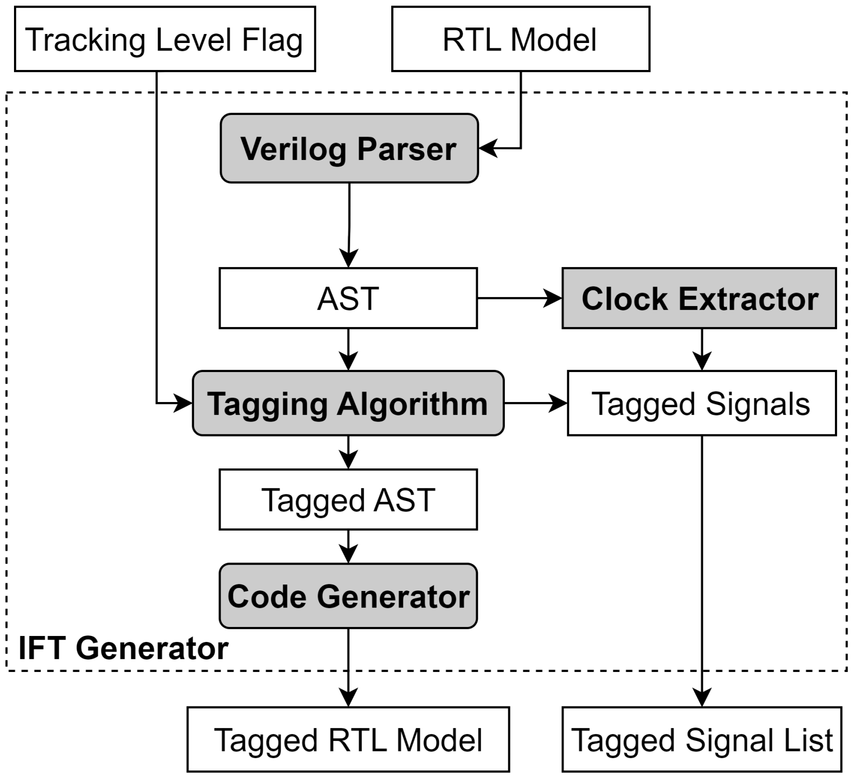


Figure 3.5: IFT Generator system flow.

3.2.2 Tagging Algorithm

Using the information flow model described by Denning [1]: $\mathcal{FM} =< \mathcal{N}, \mathcal{P}, \mathcal{SC}, \bigoplus, \rightarrow >$, we define our tagging algorithm according to the specifications in Fig. 3.6. Taking into account the risk of covert channels described in section 2.1.1, we consider any object that contains or handles information as a storage object ($\mathcal{N}$). This includes registers, ports, and wires. Similarly, we classify procedural and continuous assignments as processes ($\mathcal{P}$). We define two security classes ($\mathcal{SC}$) for the tagged signals: low (non-secured) and high (secured) and the allowed information flow ($\rightarrow$), as defined in the lattice of Fig. 2.5a. Finally, the resulting security class ($\mathcal{SC}$) is calculated using the logical disjunction operator on each of the inputs $\mathcal{SC}$s.

```
𝒩 = [input, output, inout, register, wire]
𝒫 = [assignment statements]
𝒮𝒞 = [low, high]
⨁ = logical OR
→ = [low → high, high ↛ low]
```

Figure 3.6: Information flow model definition of the IFT.

We implemented the tagging algorithm described in Alg. 1 and inspired by the work in [2,38].

The tagging algorithm receives the design's AST and the tracking level flag. Then it traverses the AST. For each module definition in the definition list of the AST, the algorithm creates the tags for all the $\mathcal{N}$ specified in Fig. 3.6. This is done by duplicating the items in the port and declaration list nodes inside the AST and renaming the tag's name with the format <name>+ "_t", with the exception of clocks and resets.

As shown in Fig. 3.3, the item list in each module can contain one of the following objects: Register, parameter, continuous assignment, instance list, or always block. If the object is a continuous assignment, the algorithm duplicates the object using the tagged signals. In the case of the instance list, each instantiation module is modified to include the tagged signals added in the previous step.

The always block can contain blocking and non-blocking substitutions, if statements, and

Algorithm 1 Tagging Algorithm

Input: AST, Tracking Level Flag $flag$.

Output: Tagged AST.

```
 1: function AST_TRAVERSER(AST, $flag$)
 2:     for module in definitions do
 3:         ports ← get_ports (module)
 4:         items ← get_parameters (module)
 5:         for each port in ports do
 6:             ports ← ports ∪ port + "_t";
 7:         for item in items do
 8:             if item is assignment (A: Y = Exp($X_i$, ... $X_j$) then
 9:                 items ← items ∪ Y_t = $X_i$ | ... | $X_j$;
10:             if item is instance then
11:                 Extend instance ports
12:             if item is Always block then
13:                 statements ← get_statements (item)
14:                 Traverse statements until assignment is found
15:                 if $flag$ == Explicit then
16:                     statements ← statements ∪ Y_t = $X_i$ | ...| $X_j$;
17:                 else if $flag$ == Conservative then
18:                     cond ← get_conditions (OP)
19:                     statements ← statements ∪ Y_t = $X_i$ | ...| $X_j$ | cond_t;
    return Tagged AST, Tagged Signals List
```

case statements. Conditional statements can contain nested conditional statements or more blocking and non-blocking substitutions. Therefore, when an item in the module definition corresponds to an always block, the algorithm must traverse the content. When the program detects a node with an information flow (procedural assignments), it inserts an assignment according to the control flow policies specified by the tracking level flag. If the flag is set to an explicit flow, the new assignment statement will include the tagged signals related to the signals in the original statement. On the other hand, if the flag is set to implicit flow, the new assignment statement will also consider the condition signals needed to reach this node.

After creating the tagged AST, we re-generate the Verilog code with the IFT logic and save it into a Verilog file that can be synthesized using Vivado. Additionally, we create a list of tagged signals along with their information (top module, tag name, clock) to ease the assertion generation in the next step.

3.2.3 Assertion Generator for IL

After the tagging algorithm and the clock extractor built the tagged signals list, the user fills in the information specifying the secure assets and non-secure outputs. The security level of each classification is given by $\mathcal{SC}$ specified in Fig. 3.6.

Algorithm 2 Assertion Generation for Information Leakage

```
Input: Design $D$, Security Assets, Outputs.
Output: Security Assertions $A$.
1: function ASSERTION GENERATION($D$, $a$)
2:     $A \leftarrow \varnothing$
3:     for $i$ in assets do
4:         for $j$ in outputs do
5:             $A \leftarrow A\cup$ assert property (assets [i] | => !outputs[j])
6:             $A \leftarrow A\cup$ assert property (assets [i] | => !outputs[j] throughout assets [i])
       return $A$
```

We set the security properties for confidentiality to detect information leaks from secure assets ($\mathcal{SC}$ = high = 1) to non-secure outputs ($\mathcal{SC}$ = low = 0). Additionally, the allowed information flows ($\rightarrow$) in Fig. 3.6 indicates that the information flow from trusted assets to untrusted outputs is forbidden.

In this work, we implemented a template-based automated assertion generator that considers the above-mentioned specifications and creates assertions that monitor secure assets, non-secure outputs, and the information flow through their $\mathcal{SC}$. The security assertions are built with all possible combinations between trusted/secure tags (assets) and untrusted/non-secure output tags (outputs), following the steps in Algorithm 2. The assertion in line 5 is adapted from the work in [2]. Additionally, we proposed the assertion in line 6, which uses the throughout operator to verify the unsecured outputs every time there is a secure asset present.

The generated assertions will be read by MBAC in the next step to generate the assertion checkers. According to the MBAC specifications, the assertions must be grouped inside an instance of a vunit module and have the clock name specified for each assertion. Therefore, we adapt the output file of the assertion generator to the template in Fig. 3.7. The vunit is instantiated with the module name and the prefix ”vu_”. If the model contains multiple modules, a different vunit is instantiated for each one.

```
vunit vu_<module> (<module>) {
assert property (@(<edge> <clk>) (asset |=> !output);
}
```

Figure 3.7: Template used for the IFT assertion generation.

3.3 IST Generator

The IST generator is composed of three functions: A Yosys adapter, an FSM reader, and an automated assertion algorithm for IST vulnerabilities. Normally, Yosys is executed manually from a command prompt by enabling the OSS-CAD environment [70]. The Yosys adapter allows our tool to automatically enter the environment and run Yosys through a synthesis script. Yosys detects and extracts the FSM of the RTL model. However, it outputs the FSM information in plain text without any data structure. The second submodule, the FSM reader, parses the text file generated by Yosys and creates a data structure that stores the relevant information. Finally, with the FSM data and the user's state specification, the automated assertion generator creates a file containing the security assertions. Since the first part of this section relies on Yosys to extract the FSM, we describe the framework before going into the details of the IST generator.

3.3.1 Yosys

Yosys is a Free and Open Source Software (FOSS) framework serving as an extensible, accessible, universal, and vendor-independent synthesis tool. It extensively supports Verilog-2005 and partially VHDL while providing a set of synthesis and optimization algorithms, all implemented in C++ [73]. It can be installed as a standalone tool from [70] or as part of the OSS CAD Suite [74], which includes other tools for RTL synthesis, formal hardware verification, place and route, and FPGA programming.

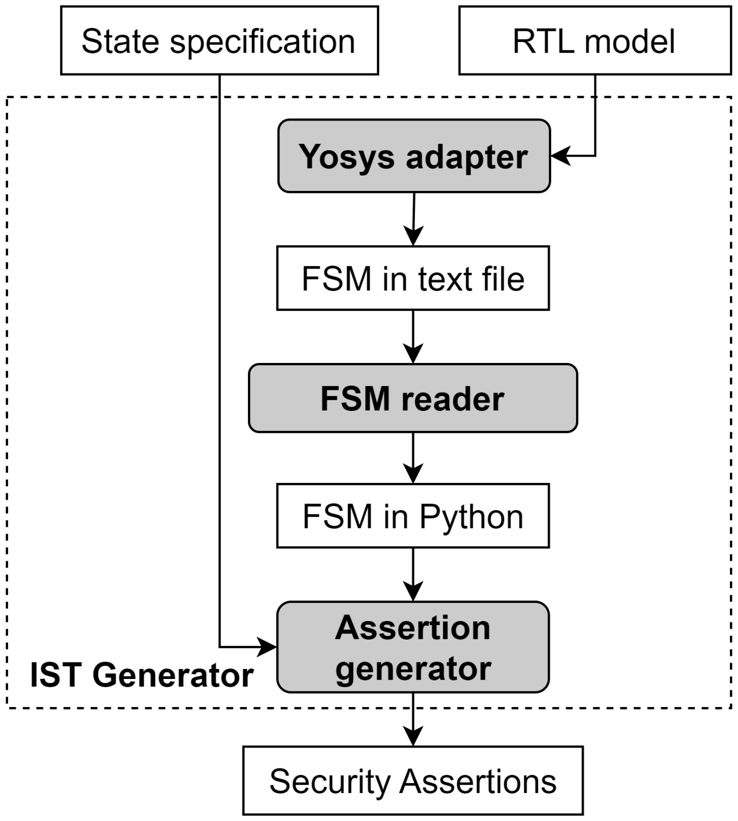


Figure 3.8: IST generator components.

A simplified Yosys data flow, extracted from [73], is shown in Fig. 3.9. In the figure, the rectangles represent program modules, and the ellipses represent internal data structures. It starts by parsing the Verilog or VHDL code and generating an AST. Then, it is passed to the AST frontend, which compiles it into a custom internal data format called RTL Intermediate Language (RTLIL). Yosys offers different commands, also called transformation or passes, to modify the RTLIL. In the end, the RTLIL is fed to the Verilog backend to generate the Verilog netlist or the RTLIL backend, which writes the RTLIL data in one of the available formats [73].

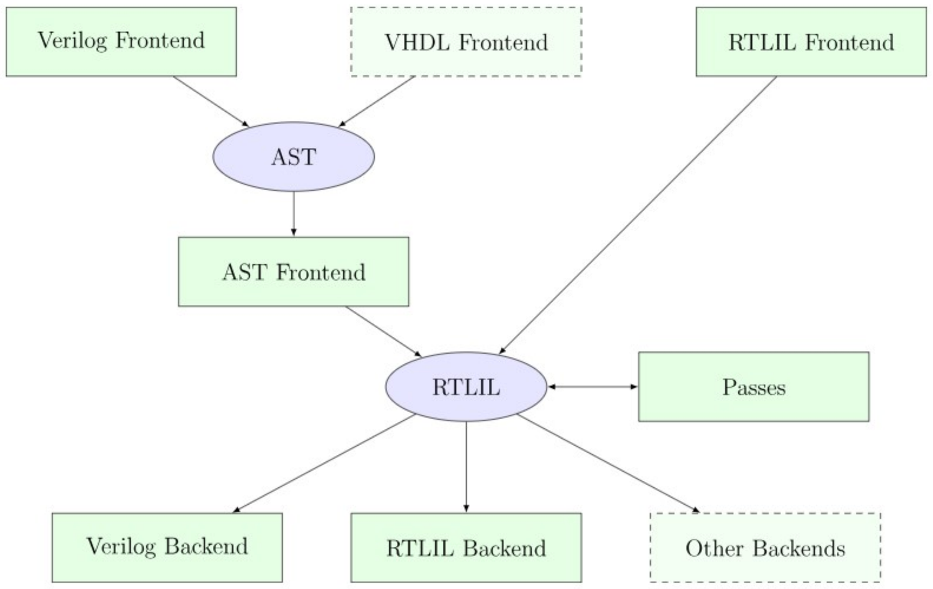


Figure 3.9: Yosys simplified data flow.

The user can call all the commands using a synthesis script containing the Yosys text commands or by using a command window and writing each command individually. Additionally, Yosys allows the user to create custom commands to transform the RTLIL or extract information from it. Detailed information on the installation, operation and commands of the OSS CAD Suite and Yosys can be found in [73].

3.3.2 Yosys Adapter

Similarly to the IFT generator, the IST generator is implemented using Python. In order to automate the generation process, we introduce the Yosys adapter submodule. The submodule activates the OSS-CAD environment that contains Yosys. From this environment, Yosys can be executed using a script via command prompts.

The script implemented for this research contains the following commands to extract the FSM from the design.

- **Read_verilog**: It loads the module from one or more Verilog files to design.
- **Hierarchy**: It checks that every module needed is included in the project and defines the top module.
- **Proc**: It calls other processing transformations. In summary, it transforms the high-level modules into multiplexers, flip-flops, and latches.
- **Memory**: It calls other memory transformations, which convert memories to word-wide DFFs and address decoders.
- **Fsm_detect**: It detects FSM by identifying the state signal. When a state signal is found, it is marked with the attributed ”fsm_encoding” used in other functions.
- **Fsm_extract**: It operates on the signals marked with the ”fsm_encoding attribute. It extracts the logic that creates the state signal and uses it to generate the control signal.
- **Fsm_info**: It prints all internal information of the extracted FSMs.
- **Tee**: It executes a command provided as an argument and writes the output into a log file.

Once Yosys reads the design, it parses it and generates the RTLIL representation. This structure is transformed and optimized by the commands previously listed until the FSM information is extracted. Yosys extracts the FSM using the method presented in [75], which analyzes the flattened gate-level netlist of the design. It identifies the FFs that contain a combinational feedback path from their output. Then it reduces the set of possible state FFs by grouping the FF controlled by the same signals.

The main limitation of Yosys is that the information about the FSM is only accessible from Yosys and it cannot be exported into any data structure (e.g., JSON File). A workaround implemented for this research was to print the FSM information into a log file in plain text. An example of the content of the log file is shown in Fig. 3.10.

3.3.3 FSM Reader

The lack of data structure in the plain text form of the FSM complicates its use for future processing. To overcome this challenge, the FSM reader submodule utilizes regular expressions to detect, extract and store any relevant information. The regular expressions take advantage of the template utilized by Yosys to present the FSM information. In Tab. 3.1, we summarize the regular expressions used and the information they gather. In particular, we extract the module name, FSM name, size, transition list, and encoding of the states. This information is saved into a data structure more suitable for assertion creation.

```
FSM '$fsm$\state$32' from module 'top':
-------------------------------------

  Information on FSM $fsm$\state$32 (\state):

  Number of input signals:    2
  Number of output signals:   5
  Number of state bits:       2

  Input signals:
    0: \reset
    1: \enable

  Output signals:
    0: $0\state[1:0] [0]
    1: $0\state[1:0] [1]
    2: $eq$test.v:22$2_Y
    3: $eq$test.v:24$3_Y
    4: $eq$test.v:26$4_Y

  State encoding:
    0:        2'00  <RESET STATE>
    1:        2'10
    2:        2'01

  Transition Table (state_in, ctrl_in, state_out, ctrl_out):
      0:      0 2'00   ->     0 5'00100
      1:      0 2'10   ->     2 5'00101
      2:      0 2'-1   ->     0 5'00100
      3:      1 2'-0   ->     0 5'10000
      4:      1 2'-1   ->     0 5'10000
      5:      2 2'-0   ->     1 5'01010
      6:      2 2'-1   ->     0 5'01000
-------------------------------------
```

Figure 3.10: FSM Information extracted using Yosys.

Table 3.1: Regular expressions used to extract FSM information.

Parameter	Regular Expression used
Module name	FSM.*from module \‘([ˆ\’]+)\’
Size	Number of state bits:\s+(\d+)
FSM name	Information on FSM [ˆ()]+ ([^)]+):
State encoding	\s+(\d+):\s+(\d+)’(\d+)\s*(?:<RESET STATE>)?
Transition list	\s+\d+:\s+(\d+)+\s+\d+’[01-]*\s+->\s+(\d+)+\s+\d+’\d+

3.3.4 Assertion Generator for IST

Similarly to the assertion generation for IL, we used a template-based approach for the assertion generation for IST. The assertion generation analyzes the FSM of the system and compares it to the protected, authorized, and illegal states declared by the user. We adopted the approach of Witharana et al. [2] to create assertions that monitor the transitions between states and verify that no transition occurs between an illegal state and a protected state. The algorithm for the assertion generation is shown in Alg. 3.

Algorithm 3 Assertion Generation for Illegal States and Transitions

```
Input: Protected States $pStates$, Authorized States $aStates$, Finite State Machine $FSM$.
Output: Security Assertions $A$.
1: function ASSERTION GENERATION($D$, $a$)
2:     $A \leftarrow \varnothing$
3:     $iStates \leftarrow$ get_Illegal_states ($pStates$, $aStates$, $FSM$)
4:     for $iS$ in $iStates$ do
5:         for $pS$ in $pStates$ do
6:             $A \leftarrow A \cup$ assert property ($iS$ |=> !$pS$)
       return $A$
```

The assertions generated by Alg. 3 are grouped by module and adapted to the MBAC specifications following the template in Fig. 3.11.

```
vunit vu_<module> (<module>) {
assert property (@(<edge> <clk>) (illegal_state |=> !protected_state);
}
```

Figure 3.11: Template used for the IST assertion generation.

3.4 Automated Assertion Checker Generator

The automated assertion checker generator in Fig. 3.12 contains three main functions: The declaration signal generator, MBAC, and the merger of RTL codes. As mentioned in section 3.2, MBAC reads a Verilog design and a file containing the assertions written in PSL or SVA.

Since MBAC specializes in generating assertion checkers from SVA and PSL assertions, it offers limited support for Verilog parsing. Advantageously, MBAC only needs to parse the signal declarations of the Verilog design to generate the assertion checkers, specifically the signal type, name, size, and range. To avoid errors produced by unsupported Verilog directives, we create a function that extracts the signal declaration from the original design.

This function is implemented using the same principle as the IFT generator. The AST is extracted from Verilog code, and every item corresponding to a signal declaration is added to the new AST (e.g., inputs, outputs, registers, wires, and parameters). Finally, the new AST is processed by the code generator function to generate the declaration file.

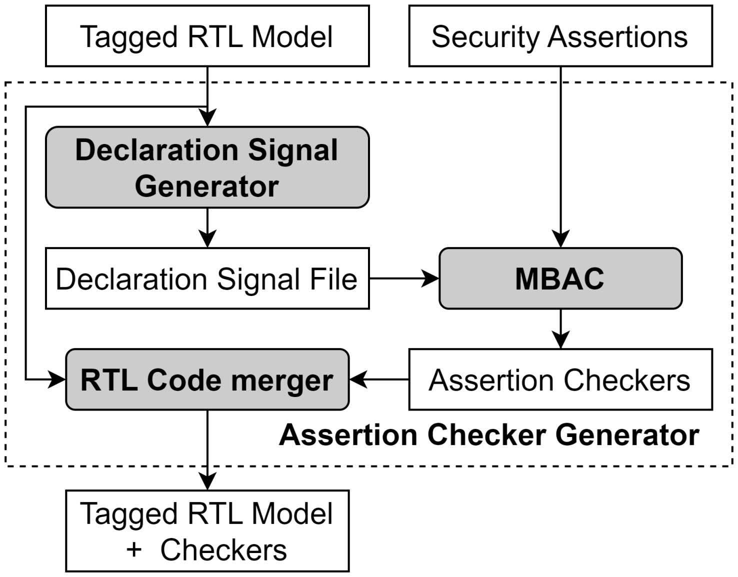


Figure 3.12: Assertion checker generation flow.

We adapted MBAC to the Python environment to receive the declaration file and the necessary commands using the subprocess management module [76]. Afterwards, MBAC converts each module and its assertions into synthesizable Verilog code and saves them into a separate file. The resulting output file is added to the project directory, and the design's source files are modified to instantiate the vu_modules created by MBAC.

3.5 Resource Utilization

As part of the literature review performed for this research, we found that research that involved information tracking flow or assertion checker generation reported the impact of the added logic in the resource utilization in terms of logic cells, slice registers, slice LUTs, slices, LUT-FF pairs, or gate equivalents. For example, GLIFT [50] measured the FPGA logic cells required in terms of LUTs. Hu et al. [66] analyzed the impact of their IFT models in terms of slice registers, slice LUTs, slices, and LUT-FF pairs. Shin et al. [67] report the number of LUTs used for their hardware-based IFT technique. Finally, Neishaburi et al. [3] evaluate the implementation of assertion-checkers clustering in terms of Gate Equivalents (number of 2-input NAND gates).

We utilize Vivado Design Suite 2021 to synthesize the Verilog designs used in this research. Afterwards, we can create a report on resource utilization. For this research, we extracted the number of slice logic and LUT-FF used by each design.

Chapter 4

Results and Discussion

In this chapter, we report the results obtained after running each module through the automated assertion checker generators. The chapter is divided into two sections. The first one presents the results related to the assertion checker generator for IL, while the second one presents the results of the assertion checker generator for IST.

The first section analyzes the impact of the IFT generator depending on the type of information flow (explicit or implicit) and compares them against the baseline design. Then, it discusses the number of assertions generated and compares them with other approaches. Afterwards, we measure the impact of our tool on the slice logic, LUT and FF quantity and compare them with the baseline designs. Finally, we evaluate the execution time of each submodule. The applicability of our tool was assessed across various Verilog projects. RS232 [12], AES-T1100 [12], CPU8080 [13], PIC16F84 [12], and VexRISCV processor [14].

In the second section, we present the results obtained by the assertion generator for IST. Then, we analyze and discuss the FPGA resources overhead produced by the IST generator on a simple FSM, RS232 [12], and CPU8080 [13]. In the end, we evaluate the execution time of the functions inside the tool.

4.1 IFT Generator

4.1.1 Design Tagging

We recreated the examples provided in [2] and shown in Fig. 4.1a. Each always block corresponds to one example from the work by Witharana et al. Note that in their research, they used one example for explicit tagging and the other one for implicit tagging, while we used both examples for the two information flow types. After running the IFT generator for explicit and implicit flows, we obtained the tagged code shown in Fig. 4.1b and Fig. 4.1c. Our tagging algorithm produced the same code as those of the original article [2] shown in Fig. 4.2 except for explicit tagging when a constant is assigned. In this case, we assigned the parameter ”LOW_TAINTED” (which has a value of 0) to imply that no secure information is flowing to the left-hand side (LHS) of the assignment (e.g., the initial state of the system after a reset), while in the example in Fig. 4.2a, they assign the value of 0.

Fig. 4.1a shows a snippet of the Verilog code without any tagging and serves as the baseline for comparison with the tagged versions. In Fig. 4.1b, we present the output code after applying explicit tagging, where an assignment operation propagates the $\mathcal{SC}$ of the input signals to the output signal. For the implicit approach in Fig. 4.1c, the tool also took into account the $\mathcal{SC}$ of the objects in the conditional statements, e.g., the SC of ”a” in the if statement (a_t) flows to c_t.

```
always @(posedge CLK) begin
   if(a) begin
     c <= b;
  end else begin
     c <= 0;
  end
end
always @(posedge CLK) begin
     a = s;
     b <= a;
end
```

(a) Snippet of Verilog code.

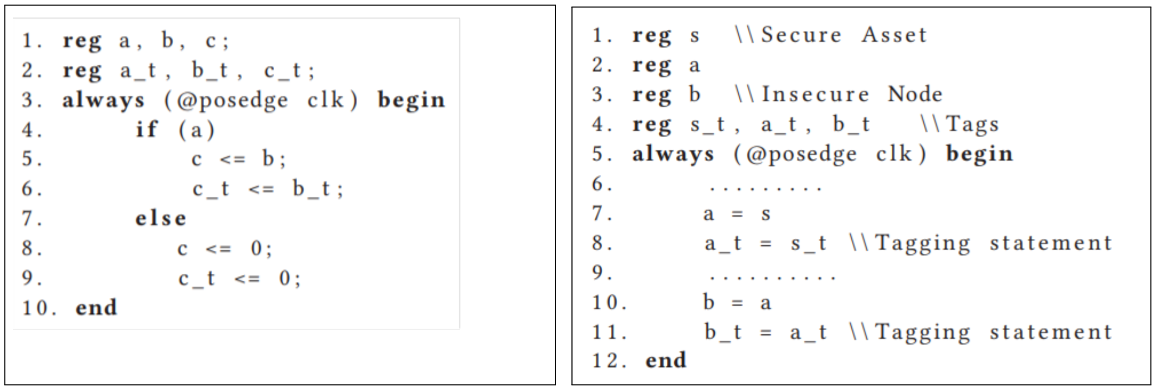
```
always @(posedge CLK) begin
   if(a) begin
     c <= b;
     c_t <= b_t;
  end else begin
     c <= 0;
     c_t <= LOW_TAINTED;
  end
end
always @(posedge CLK) begin
  a = s;
  b <= a;
  a_t = s_t;
  b_t <= a_t;
end
```

(b) Explicit tagging.

```
always @(posedge CLK) begin
   if(a) begin
     c <= b;
     c_t <= b_t | a_t;
  end else begin
     c <= 0;
     c_t <= a_t;
  end
end
always @(posedge CLK) begin
  a = s;
  b <= a;
  a_t = s_t;
  b_t <= a_t;
end
```

(c) Implicit tagging.

Figure 4.1: Examples of tagging logic.



```
reg a, b, c;
reg a_t, b_t, c_t;
always (@posedge clk) begin
      if (a)
            c <= b;
            c_t <= b_t;
      else
          c <= 0;
          c_t <= 0;
end
```

(a) Explicit tagging.

```
reg s     \\Secure Asset
reg a
reg b     \\Insecure Node
reg s_t, a_t, b_t    \\Tags
always (@posedge clk) begin
      .........
      a = s
      a_t = s_t \\Tagging statement
      ..........
      b = a
      b_t = a_t \\Tagging statement
end
```

(b) Implicit tagging.

Figure 4.2: Examples of tagged code extracted from [2]

We evaluated the resource utilization of the two tagging approaches and compared them with the resource utilization of the baseline (original Verilog code). We ruled out the influence of the PyVerilog parser and code generator on resource utilization by parsing the design into an AST and regenerating the original code without modifying or adding tagging logic. We used the Vivado Design Suite to synthesize the RTL code and generate a resource utilization report to measure the slice logic, LUT, and FF utilization.

The results shown in Tab. 4.1 provide information on the effects of our tagging methodology on resource utilization. For each design, implicit tagging utilized more slice logic resources with the exemption of the AES-T1100 core. The encryption core used the same amount of slice logic for explicit tagging and implicit tagging. An inspection of the source code showed the absence of conditional statements, as the operations are logical and arithmetical, creating a similar tagged design for both tagging methods. The slice logic utilization varied per design but increased overall for every design.

We report a maximum increase of slice logic utilization of up to 10.58% and 33.33% in the use of slice logic for explicit and implicit tagging, respectively. The most significant overhead occurs in smaller designs such as the RS232 and PIC16F84 and a smaller increase for larger designs, suggesting a downward trend for more complex structures.

Table 4.1: Slice logic utilization per tagging technique.

Name	Baseline	Explicit	Logic Increase	Implicit	Logic Increase
RS232	156	167	7.05%	208	33.33%
AES-T1100	10964	10969	0.05%	10969	0.05%
CPU8080	1511	1598	5.76%	1638	8.41%
PIC16F84	718	794	10.58%	946	31.75%
VexRISCV	1222	1243	1.72%	1248	2.13%

In addition to the slice logic utilization, we registered the number of LUTs and FFs utilized by the designs. The results are summarized in Tab. 4.2. Similarly to resource utilization, implicit tagging produced a larger overhead compared to explicit tagging. In the case of the PIC16F84, the implicit tagging showed the highest percentage increase in LUT and FF utilization. However, when calculating the absolute difference between the baseline and the implicit tagging, the design showed an increase of 116 LUTs and 81 FF. Since current FPGAs such as the ARTY A7-100 contain LUTs and FFs in the order of the thousands [77], the increase is neglectable.

Table 4.2: LUTs and FFs utilization per tagging technique.

	Baseline		Explicit		Logic Increase		Implicit		Logic Increase	
Name	LUT	FF	LUT	FF	LUT	FF	LUT	FF	LUT	FF
RS232	64	73	66	83	3.13%	9.21%	82	104	7.81%	9.21%
AES-T1100	3185	5776	4783	4650	0.06%	0.03%	4783	4650	0.06%	0.03%
CPU8080	1016	243	1061	269	5.04%	11.11%	1108	276	7.17%	13.58%
PIC16F84	368	270	394	299	8.00%	10.58%	477	356	30.93%	29.56%
VexRISCV	556	555	557	576	0.18%	3.78%	574	593	0.18%	4.86%

4.1.2 Assertion Generation

From the example in Fig. 4.1, we generated an asset list and declared signal b as non-secure and signal s as a secure asset. After running the automated assertion generator, we obtained the assertions declaration below. The first one corresponds to the assertion proposed in [2], and the second one is the assertion proposed in this work. Since we need to group the assertions per module to generate the assertion checkers, we added information regarding the vu_unit module to the assertion file.

```
vunit vu_top( top) {

assert property ( @(posedge CLK) s_t |=> !b_t );

assert property ( @(posedge CLK) s_t |=> !b_t throughout s_t);

}
```

Figure 4.3: Assertion generated for example code.

We used the automated tool to generate assertions for the RS232-T100, PIC16F84-T400, AES-T1100, CPU8080, and VexRISCV designs and summarize the results in Tab. 4.3. The table includes the number of secure assets and insecure outputs declared by the user for each module. The fourth column shows the results reported in [2]. The fifth column shows the number of assertions generated using the assertion template proposed in [2]. We found the same number of assertions generated for each module except for the PIC16F84 module. The only difference between the methodology of [2] and ours is that we exclude the signals related to clocks and resets. Additionally, we used our tool in VexRISCV, a RISC-V ISA implementation, which was not included in the work by Witharana et al. The sixth column reports the number of assertions using the template mentioned before with the addition of a second assertion template.

Table 4.3: Assertion generation for IL.

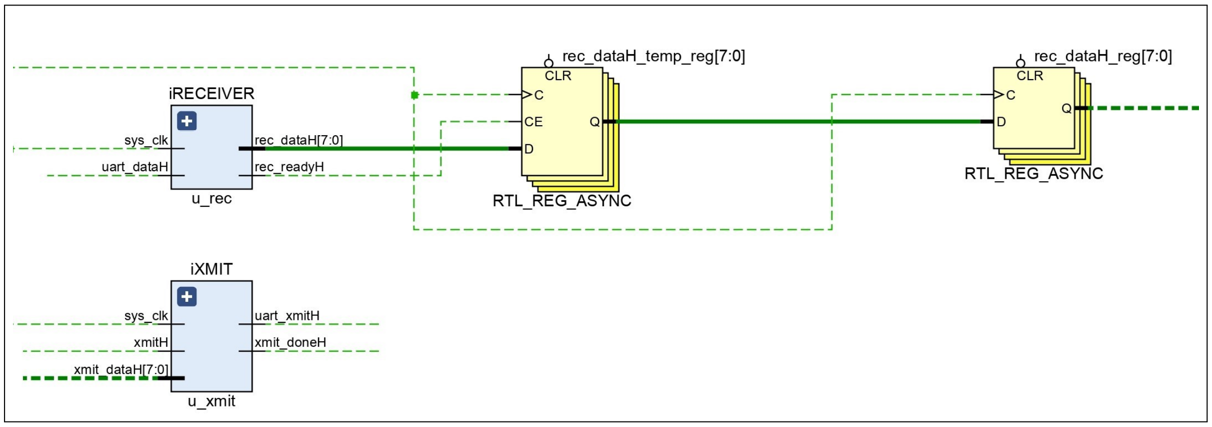
Name	Secure Assets	Unsecure Outputs	Assertions in [2]	Assertions	Enhanced Assertions
RS232	1	4	4	4	8
AES-T1100	1	1	1	1	2
CPU8080	1	6	6	6	12
PIC16F84	1	17	18	17	34
VexRISCV	2	7	N/A	14	28

The number of assertions generated depends on the combinations of secure assets and non-secure outputs. Since the number of secure assets tends to be small, the number of assertions is proportional to the number of non-secure outputs specified by the user. For this research, we only monitored the top module's output signals. However, other objects can be included besides output signals, and there can be more than one secure asset, which can rapidly increase the number of assertions generated.

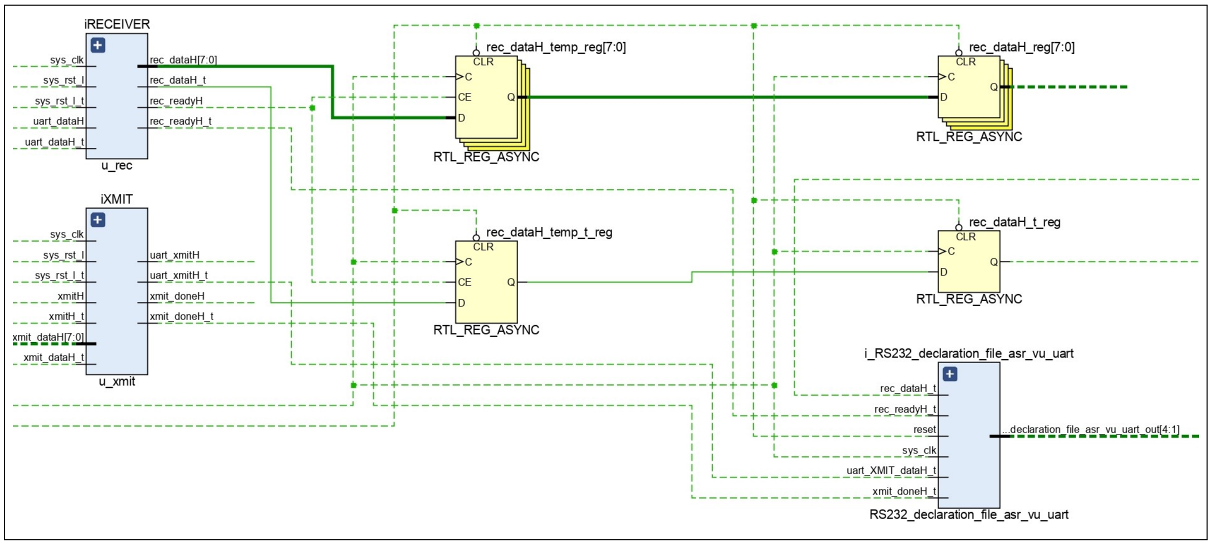
4.1.3 Automated Assertion Checker Generation

The output file of the tool proposed in this work contains a Verilog design that integrates the original code, IFT, and hardware assertion checkers that monitor the tags of the secure assets. The resulting code was synthesized using the Vivado Design Suite, from which we generated the resource utilization report.

We provide an example of the final system following the flow in Fig. 3.12. The schematic of the RS232 core in Fig. 4.4 (a) consists of two submodules (iReceiver and iXMIT) and two groups of registers. Fig. 4.4 (b) shows the two submodules with the addition of the tagged signals, two shadow registers equivalent to the original register groups, and the vu_unit created by MBAC that contains the hardware assertion checkers and the output signal reporting the state of the assertions.



(a) Baseline module.



(b) Module with explicit tagging merged with assertions checkers.

Figure 4.4: RS232 module circuit diagram before and after introducing the assertion checkers.

The process was repeated for each module assessed in this thesis, and the results regarding the utilization of FPGA resources are summarized in Tab. 4.4 and 4.5. The table presents the slice logic utilization for each tagging process with and without the addition of assertion checkers. The second and fifth columns show the slice logic utilization for the designs with explicit and implicit tagging, respectively. The third column shows the utilization after adding the assertion checker to the explicitly tagged design, while the sixth column shows the results for the implicit tagging with assertions. Similar to the results from Tab. 4.1, the biggest resource utilization happened in the smallest designs. In the case of the PIC16F84, the large resource increase can also be explained by the number of assertions used to create the assertion checkers.

Table 4.4: Slice logic utilization after assertion checker integration for IL.

Name	Explicit tagging	Explicit with Assertions	Logic increase	Implicit tagging	Implicit with Assertions	Logic increase
RS232	167	189	13.17%	208	221	6.25%
AES-T1100	10969	10977	0.07%	10969	10977	0.07%
CPU8080	1598	1637	2.44%	1638	1685	2.87%
PIC16F84	794	898	13.10%	946	1055	11.52%
VexRISCV	1243	1323	6.44%	1248	1333	6.81%

Table 4.5: LUTs and FFs utilization after assertion checker integration for IL.

	Baseline		Explicit with Assertions		Logic Increase		Implicit with Assertions		Logic Increase	
Name	LUT	FF	LUT	FF	LUT	FF	LUT	FF	LUT	FF
RS232	64	76	72	99	12.50%	30.26%	85	112	32.81%	47.37%
AES-T1100	3185	5776	3190	5782	0.16%	0.10%	3190	5782	0.16%	0.10%
CPU8080	1032	243	1097	293	6.30%	20.58%	1121	300	8.62%	23.46%
PIC16F84	375	274	431	371	14.93%	35.40%	518	423	38.13%	54.38%
VexRISCV	556	555	568	632	2.16%	13.87%	572	638	2.88%	14.95%

The graph in Fig. 4.5 shows the execution time for each submodule of the automated tool. The IFT generator and the assertion checker generator used most of the execution time, while the time spent in the assertion generation was negligible (around 0.004 s on average). A closer analysis of the two submodules, shown in Fig. 4.6, indicated that they spent most of the execution time in the parser, clock extractor, and declaration file generator functions. These functions have in common that they use the PyVerilog libraries to extract or generate information.

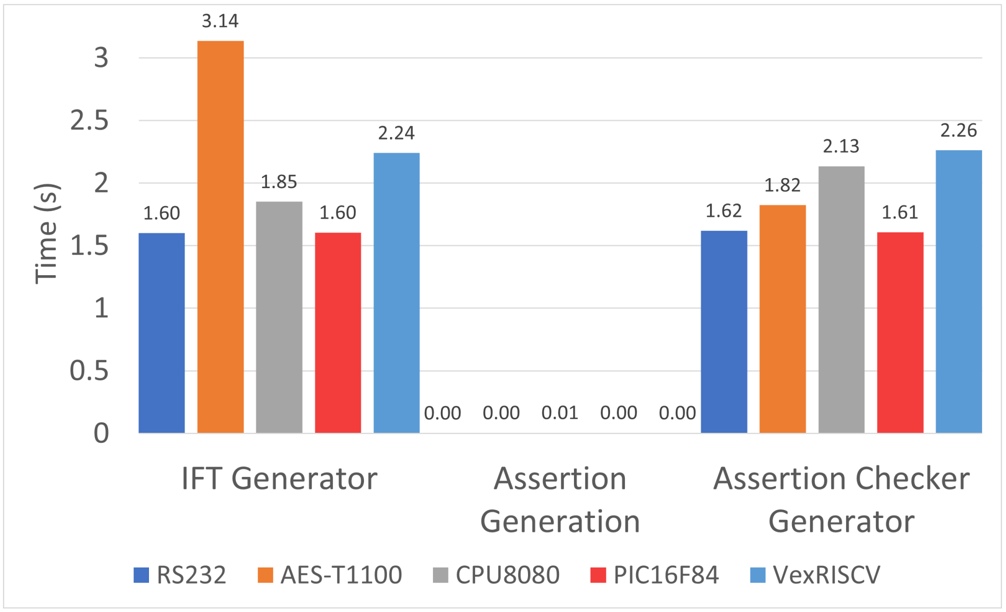
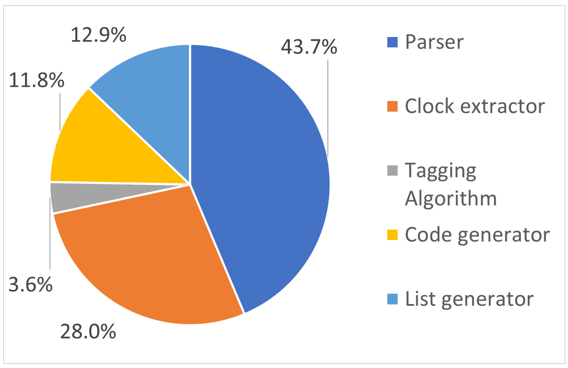
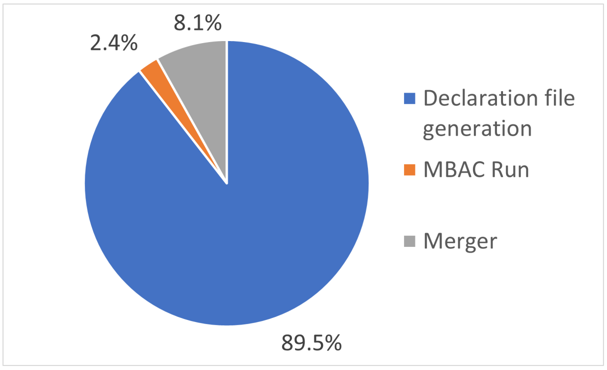


Figure 4.5: Execution time of the IL assertion checker generator.



(a) IFT generator.



(b) Assertion checker generator.

Figure 4.6: Execution time of IFT generator and assertion checker generator submodules.

4.2 IST Generator

4.2.1 Assertion Generation for IST

Using the assertion generator described in Section 3.3, we generated a list of assertions for three designs: a simple FSM, RS232-T100, and CPU8080. Note that the other Verilog designs were not included since Yosys did not detect an FSM. The behaviour is consistent with the one reported in [2]. We compared our results, shown in Tab. 4.6, with the ones obtained from Witharana et al. [2] and found a discrepancy for both designs. Different factors can explain these differences. First, the specification of protected and authorized states is open to the design criteria, and this specification will impact the final number of assertions generated. Another factor is that Yosys automatically detects the FSMs at the gate level and follows the detection criteria specified in [78]. For example, the RS232 design has two similar modules (iReceiver and iXMIT), but Yosys only detected an FSM in one of them.

Table 4.6: Assertion generation for IST.

Name	Protected States	Authorized States	FSM size in bits	Assertions in [2]	Assertions
Demo code	1	1	2	N/A	2
RS232	1	1	3	15	6
CPU8080	1	1	5	32	62

4.2.2 Automated Assertion Checker Generation

Once the assertions were transformed into assertion checkers and merged into the design, we synthesized each design and generated a logic slice utilization report for the baseline design and the design with the assertions added. The logic slice utilized for baseline design and design with assertions is shown in Tab. 4.7. We found that, on average, the slice logic increased by 13.35%. The resource utilization in terms of slice logic was approximately the same for the three designs despite the size and number of assertions used. However, if we focus on the LUT and FF resource utilization described in Tab. 4.8, we found a smaller usage percentage of LUTs for the largest design (CPU8080) but a larger utilization of FF, possibly due to the number of assertions used for this design.

The execution time analysis shown in Fig. 4.7 shows that most of the time was spent in the Yosys adapter submodule. Since we are using Python to open and execute the command prompt to activate the OSS-CAD environment and send the Yosys instructions, we can

Table 4.7: Slice logic utilization after assertion checker integration for IST.

Name	Baseline	Baseline with Assertions	Logic increase
Demo code	64	73	14.06%
RS232	156	179	14.74%
CPU8080	1490	1678	12.62%

Table 4.8: LUTs and FFs utilization after assertion checker integration for IST.

	Baseline		With Assertions		Overhead	
Name	LUT	FF	LUT	FF	LUT	FF
FSM	27	27	29	31	7.41%	14.81%
RS232	64	76	69	88	7.81%	15.79%
CPU8080	1032	243	1080	370	4.65%	52.26%

expect a large time overhead.

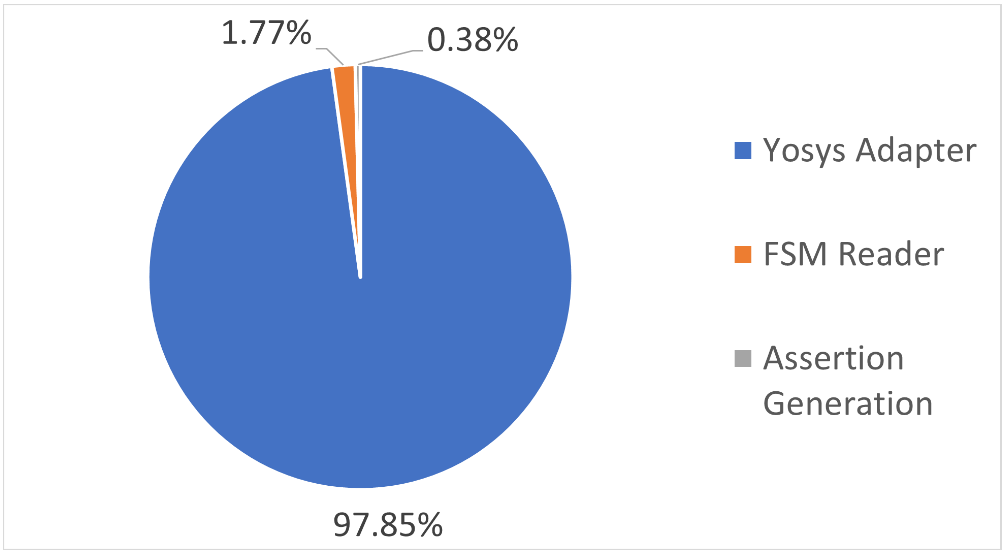


Figure 4.7: Execution time of the IST assertion checker generator.

Chapter 5

Conclusions and Future Work

In this work, we proposed two tools that integrate information flow tracking, FSM detection and extraction, security assertion generation, and assertion checker generation to automate the generation of hardware assertion checkers specialized in the security verification of information leakage vulnerabilities and illegal states and transitions. The tagging algorithms described and implemented in this work produced results similar to those described in previous studies. We found that the implicit tagging methodology produced a bigger overhead in terms of slice logic utilization than the explicit tagging methodology, up to a 33.33% increase from the baseline design when using implicit tagging and up to 10.58% increase when using explicit tagging. We demonstrate the correctness of our assertion generation methodology for IL by comparing the results with current approaches [2]. In the case of assertion generation for IST, our results diverged from similar approaches, mainly due to the FSM detection by Yosys. For the IFT generator tool, we observed that the assertion checkers generated by MBAC carried less than 7% FPGA resource overhead for large designs and up to 13.17% for smaller designs. In the case of the IST generator tool, we observed a consistent slice logic increase of 13.35% on average. In general, our tool has a moderate impact on resource utilization, especially when using the implicit flow tagging methodology. The largest increase in LUT utilization was 38% compared to the baseline design.

Additionally, we analyzed the execution time of each submodule and relevant functions. We found that most of the time was spent by PyVerilog and the communication overhead produced by the Yosys adapter, while the core functions and tools (tagging algorithm, assertion generation, and MBAC) utilized a fraction of the total time. Our method integrates different third-party tools that verify the inputs being supplied. For example, PyVerilog and Yosys check that the Verilog design is correctly written, MBAC verifies the information produced by the declaration file generator and the assertion generator, and Vivado verifies that the assertion checkers and design tagging outputs comply with the standards and reports any design errors present in the RTL code.

Future work

This research mainly focused on information leakage and illegal states and transition vulnerabilities. However, as mentioned in the Background section, there are other vulnerabilities that also pose a risk to hardware designs. In the future, the tools and methodology presented in this research can be tailored to the specific mechanics of each vulnerability. This work focused on integrating different subsystems, quantifying the impact of assertions created and resources overhead. However, future research should focus on performance analysis to evaluate the design in terms of power consumption and delay parameters. Additionally, this work was built over Python and relied heavily on the PyVerilog toolkit. In the future, other languages and hardware synthesis tools could be considered to benefit from their features and improve execution time. For example, Yosys, written in C++), is a well-maintained open-source project in constant improvement that already contains modules to parse, optimize, verify, simulate, and synthesize Verilog and VHDL projects. Lastly, adopting techniques to debug hardware assertion checkers could benefit the implementation of hardware assertion checkers in the final stages of the design cycle. For example, the use of assertion clusters for debugging [3] has been explored for post-silicon functional verification.

Bibliography

[1] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[2] H. Witharana, A. Jayasena, A. Whigham, and P. Mishra, "Automated generation of security assertions for rtl models," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 19, no. 1, pp. 1–27, 2023.

[3] M. H. Neishaburi and Z. Zilic, "An infrastructure for debug using clusters of assertion-checkers," *Microelectronics Reliability*, vol. 52, no. 11, pp. 2781–2798, 2012.

[4] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[5] ——, "2018 fpga functional verification trends," in *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 2018, pp. 40–45.

[6] ——, "Wilson research group functional verification study: Ic/asic functional verification trend report," *White Paper. Wilson Research Group and Mentor, A Siemens Business*, 2020.

[7] ——, "Wilson research group fpga functional verification trends," *White Paper. Wilson Research Group and Mentor, A Siemens Business*, 2022.

[8] ——, "2022 wilson research group ic/asic functional verification trends," *White Paper. Wilson Research Group and Mentor, A Siemens Business*, 2022.

[9] L. Liu and S. Vasudevan, "Automatic generation of system level assertions from transaction level models," *Journal of Electronic Testing*, vol. 29, pp. 669–684, 2013.

[10] S. Hertz, D. Pal, S. Offenberger, and S. Vasudevan, "A figure of merit for assertions in verification," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 675–680.

[11] Y. Lyu and P. Mishra, "System-on-chip security assertions," *arXiv preprint arXiv:2001.06719*, 2020.

[12] 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, pp. 85–102, 2017.

[13] "Open cores," 2021, accessed: 2023-08-06. [Online]. Available: https://opencores.org/

[14] "Vexriscv," 2022, accessed: 2023-08-03. [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[15] F. Ghenassia, *Transaction level modeling with SystemC*. Springer, 2005.

[16] D. Thomas and P. Moorby, *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[17] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs: For System-on-a-chip Designs*. Springer Science & Business Media, 2002.

[18] H. Witharana, Y. Lyu, and P. Mishra, "Directed test generation for activation of security assertions in rtl models," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 4, pp. 1–28, 2021.

[19] Z. Wang, "Information leakage due to cache and processor architectures," Ph.D. dissertation, Princeton University, 2012.

[20] 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.

[21] M. Steil, "17 mistakes microsoft made in the xbox security system," in *22nd Chaos Communication Congress*, 2005.

[22] F. Farahmandi and P. Mishra, "Fsm anomaly detection using formal analysis," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 313–320.

[23] B. Sunar, G. Gaubatz, and E. Savas, "Sequential circuit design for embedded cryptographic applications resilient to adversarial faults," *IEEE Transactions on Computers*, vol. 57, no. 1, pp. 126–138, 2007.

[24] N. Pundir, S. Aftabjahani, R. Cammarota, M. Tehranipoor, and F. Farahmandi, "Analyzing security vulnerabilities induced by high-level synthesis," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 3, pp. 1–22, 2022.

[25] S. Skorobogatov and C. Woods, "Breakthrough silicon scanning discovers backdoor in military chip," in *Cryptographic Hardware and Embedded Systems–CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings 14*. Springer, 2012, pp. 23–40.

[26] W. C. Barker, "Guideline for identifying an information system as a national security system," National Institute of Standards and Technology, Tech. Rep., 2003.

[27] Synopsys. (2023) Equivalence checking. [Online]. Available: https://www.synopsys.com/glossary/what-is-equivalence-checking.html

[28] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, 1997.

[29] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*. Springer, 2011, pp. 1–30.

[30] M. Loghi, T. Margaria, G. Pravadelli, and B. Steffen, "Dynamic and formal verification of embedded systems: A comparative survey," *International Journal of Parallel Programming*, vol. 33, pp. 585–611, 2005.

[31] T. art of verification. (2021) Direct testing vs constraint random verification. [Online]. Available: https://www.theartofverification.com/directed-testing-vs-constraint-random-verification/

[32] U. Farooq and H. Mehrez, "Pre-silicon verification using multi-fpga platforms: A review," *Journal of Electronic Testing*, vol. 37, no. 1, pp. 7–24, 2021.

[33] Z. Ren and H. Al-Asaad, "Overview of assertion-based verification and its applications," in *Int'l Conf. Embedded Systems, Cyber-physical Systems, & Applications*, 2016.

[34] M. Boulé and Z. Zilic, *Generating hardware assertion checkers*. Springer, 2008.

[35] M. Neishabouri, "Debug instrumentations and fault-tolerant techniques for on-chip networks," Ph.D. dissertation, McGill University, 2014.

[36] D. A. S. Committee *et al.*, "Ieee standard for systemverilog unified hardware design, specification, and verification language standard ieee 1800," *http://www. edastds. org/sv/*, 2005.

[37] IEEE, "Ieee standard for property specification language (psl)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010.

[38] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1691–1696.

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

[40] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 17–30, 2005.

[41] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[42] J. Bacon, D. Eyers, T. F.-M. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch, "Information flow control for secure cloud computing," *IEEE Transactions on network and Service Management*, vol. 11, no. 1, pp. 76–89, 2014.

[43] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 135–148.

[44] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 2004, pp. 243–254.

[45] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 109–120, 2011.

[46] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 97–112.

[47] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *Acm Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[48] M.-M. Bidmeshki and Y. Makris, "Toward automatic proof generation for information flow policies in third-party hardware ip," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015, pp. 163–168.

[49] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Theoretical analysis of gate level information flow tracking," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 244–247.

[50] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, 2009, pp. 109–120.

[51] C. Wolf, "Yosys open synthesis suite," 2016, accessed: 2023-08-01.

[52] F. Solt, B. Gras, and K. Razavi, "CellIFT: Leveraging cells for scalable and precise dynamic information flow tracking in RTL," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2549–2566.

[53] A. Ardeshiricham, *Verification and Synthesis of Information Flow Secure Hardware Designs*. University of California, San Diego, 2020.

[54] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "Iodine: a tool to automatically infer dynamic invariants for hardware designs," in *Proceedings of the 42nd annual Design Automation Conference*, 2005, pp. 775–778.

[55] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke, "Automatic generation of complex properties for hardware designs," in *Proceedings of the conference on Design, automation and test in Europe*, 2008, pp. 545–548.

[56] A. Hekmatpour and A. Salehi, "Block-based schema-driven assertion generation for functional verification," in *14th Asian Test Symposium (ATS'05)*. IEEE, 2005, pp. 34–39.

[57] D. Sheridan, L. Liu, H. Kim, and S. Vasudevan, "A coverage guided mining approach for automatic generation of succinct assertions," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. IEEE, 2014, pp. 68–73.

[58] A. Danese, N. D. Riva, and G. Pravadelli, "A-team: Automatic template-based assertion miner," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[59] Y. Li, W. Wu, L. Hou, and H. Cheng, "A study on the assertion-based verification of digital ic," in *2009 Second International Conference on Information and Computing Science*, vol. 2. IEEE, 2009, pp. 25–28.

[60] B. Turumella and M. Sharma, "Assertion-based verification of a 32 thread sparc™ cmt microprocessor," in *Proceedings of the 45th annual Design Automation Conference*, 2008, pp. 256–261.

[61] F. Haque and J. Michelson, *Art of Verification with VERA*. Verification Central, 2001.

[62] M. Eslami, T. Ghasempouri, and S. Pagliarini, "Reusing verification assertions as security checkers for hardware trojan detection," in *2022 23rd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2022, pp. 1–6.

[63] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–33, 2022.

[64] J. Curreri, G. Stitt, and A. D. George, "High-level synthesis techniques for in-circuit assertion-based verification," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–8.

[65] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil, "Synthesis of synchronous assertions with guarded atomic actions," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05.* IEEE, 2005, pp. 15–24.

[66] 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)*. ACM, 2018, pp. 1–8.

[67] J. Shin, H. Zhang, J. Lee, I. Heo, Y.-Y. Chen, R. Lee, and Y. Paek, "A hardware-based technique for efficient implicit information flow tracking," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–7.

[68] L. Liu, D. Sheridan, V. Athavale, and S. Vasudevan, "Automatic generation of assertions from system level design using data mining," in *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*. IEEE, 2011, pp. 191–200.

[69] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, ser. Lecture Notes

in Computer Science, vol. 9040. Springer International Publishing, Apr 2015, pp. 451–460. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16214-0_42

[70] YosysHQ. (2023) Yosys: Yosys open synthesis suite. [Online]. Available: https://github.com/YosysHQ/yosys

[71] M. S. Rahman, R. Guo, H. M. Kamali, F. Rahman, F. Farahmandi, and M. Tehranipoor, "Retrustfsm: Toward rtl hardware obfuscation-a hybrid fsm approach," *IEEE Access*, vol. 11, pp. 19 741–19 761, 2023.

[72] PyHDI. (2022) Pyverilog. [Online]. Available: https://github.com/PyHDI/Pyverilog

[73] YosysHQ. (2022) Yosys manual. [Online]. Available: https://yosyshq.readthedocs.io/projects/yosys/en/latest/

[74] ——. (2023) Oss cad suite build. [Online]. Available: https://github.com/YosysHQ/oss-cad-suite-build#installation

[75] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren, "A highly efficient method for extracting fsms from flattened gate-level netlist," in *Proceedings of 2010 IEEE international symposium on circuits and systems.* IEEE, 2010, pp. 2610–2613.

[76] P. S. Foundation. (2023) Subprocess management. [Online]. Available: https://docs.python.org/3/library/subprocess.html

[77] *XA Artix-7 FPGAs Data Sheet: Overview* , Xilinx, 2017, v1.3. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds197-xa-artix7-overview

[78] "Yosys manual: Optimizations," 2021, accessed: 2023-08-12. [Online]. Available: https://yosyshq.readthedocs.io/projects/yosys/en/latest/CHAPTER_Optimize.html#fsm-extraction-and-encoding