IFV: Information Flow Verification at the Pre-silicon Stage Utilizing Static-Formal Methodology

Rasheed Kibria, Farimah Farahmandi, and Mark Tehranipoor
Department of Electrical and Computer Engineering, University of Florida, Gainesville, Florida
Email: rasheed.kibria@ufl.edu, farimah@ece.ufl.edu, tehranipoor@ece.ufl.edu

***Abstract*—Modern system-on-chips (SoCs) are becoming prone to numerous security vulnerabilities due to their ever-growing complexity and size. Therefore, a comprehensive security verification framework is needed at the very early stage of the SoC design lifecycle. The datapath of a complex SoC design may be vulnerable to information leakage and data integrity issues. The designers might be unaware of hidden information flow paths present in a particular SoC design at the pre-silicon stage, which can eventually lead to severe data breaches. Hence, it is crucial to develop a novel framework that comprehensively identifies the presence of such paths. Moreover, novel mathematical metrics need to be formulated to perform an exhaustive quantitative assessment of the detected information leakage paths. It will assist designers in quantifying the security risk level associated with these data propagation paths, ultimately making them aware of the potential implications of these leakage paths. In this paper, we propose an information flow verification framework that utilizes a combination of static and formal methodologies to identify information flow paths based on a mathematical metric for quantifying the security risk level of the detected paths. Our experiments across numerous open-source designs, varying in size and complexity, demonstrate the efficacy of the proposed framework for identifying severe information leakage and data integrity issues at the pre-silicon stage of the design lifecycle.**

***Index Terms*—Static verification, Formal methods, Information security, SoC security verification**

I. Introduction

System-on-Chips (SoCs) serve as the computational backbone of modern *Internet of Things (IoT)* platforms, which facilitate a wide range of applications. A typical SoC might contain tens of *Intellectual Property (IP)* cores, such as analog, memory, digital, reconfigurable fabrics, etc. These IP cores have distinct and diverse functionalities, complexities, and sizes, which may present unique security issues. Such blocks might process sensitive information and be exposed to a broad spectrum of attacks. Moreover, new security vulnerabilities can emerge from the complex interactions between the IP cores present in an SoC under a real-world workload. Consequently, a comprehensive security verification framework is required at the pre-silicon stage of the design lifecycle to ensure the security and trustworthiness of SoCs before fabrication and deployment in modern systems. The design of secure SoCs is a formidable challenge, since security vulnerabilities can be introduced at multiple stages of the complex SoC design lifecycle. In general, SoC security vulnerabilities can be categorized as leakage of sensitive information, data integrity violations, loss of access control, side-channel leakage, insertion of hidden and malicious functionalities, susceptibility to fault-injection attacks, and exploitation of *Design for Test (DfT)* and *Design for Debug (DfD)* infrastructures [1]–[6]. Such vulnerabilities may arise unintentionally due to oversights in the SoC design process or a lack of security awareness among designers. Additionally, state-of-the-art automated *Computer-Aided Design (CAD)* tools can inadvertently introduce new vulnerabilities into the SoC implementations [2], [8].

The *Very-Large-Scale Integration (VLSI)* design industry traditionally dedicates a tremendous amount of time and resources to verifying SoC designs to ensure functional correctness and meet *Power, Performance, and Area (PPA)* requirements. With continuous technology scaling and the integration of numerous complex IP cores, SoCs have grown substantially in size and complexity. Hence, performing functional verification and identifying all design issues in a single phase has become exceptionally challenging and infeasible. Similarly, verification for security assurance has become even more challenging [3], [7]. The diversity of threat vectors, coupled with the absence of standardized security metrics and rules, forces verification engineers to rely on manual review and ad hoc analysis techniques. However, such manual efforts are inherently inefficient, prone to errors, subjective, and non-scalable [18]. In addition, verification engineers are often unaware of security vulnerabilities at early stages of the SoC design lifecycle, particularly at the *Register-Transfer Level (RTL)* and post-synthesis abstractions, where many critical vulnerabilities might originate [3], [8]. Early detection and mitigation of these vulnerabilities are vital, since the cost and effort of fixing security flaws increase dramatically in later stages of the SoC design lifecycle, typically increasing by an order of magnitude after each phase [19]. Therefore, incorporating a security-aware verification flow early in the design lifecycle is crucial for modern SoC designs.

Verifying the datapath of an SoC is a vital component of SoC security validation, since the compromise of the datapath can compromise the security of the entire system. The datapath of an SoC design might be susceptible to fault-injection attacks and may exhibit inherent information leakage or data integrity vulnerabilities [9], [21]. Furthermore, it is essential to develop mathematical metrics to assess the risk of information leakage quantitatively. While the proposed approaches in [21]–[26] employ a variety of quantitative metrics to evaluate leakage and related security risks, these metrics cannot generally capture the dynamic propagation of sensitive information along specific datapath routes. However, there exist substantial limitations in the existing models. For example, the graph-based models [21] and hyperflow graphs [24] cannot capture dynamic interactions. Similarly, some mathematical metrics have been proposed to address min-entropy leakage [25], as well as confidentiality and corruptibility [26]. Such metrics overlook critical spatial characteristics of sensitive data

propagation in the datapath. As a result, numerous security vulnerabilities might not be appropriately addressed. In this paper, we propose a framework called **I**nformation **F**low **V**erifier (IFV) to effectively and quickly detect information flow paths between the source and destination points at the bit level and prove a novel mathematical metric utilizing formal verification. This metric can be used to calculate the security risk level for leakage quantitatively. Specifically, our contributions in this paper are as follows-

- Development of an automated framework named *IFV* to detect information flow precisely at the bit level in the pre-silicon stage of the SoC design lifecycle;
- Formulation of a mathematical metric to assess the security risk level for information leakage quantitatively;
- Demonstration of the effectiveness of *IFV* on 12 open-source benchmarks from [11]–[13] and other sources varying in functionality, complexity, and size.

The rest of the paper is organized as follows. Section II presents the preliminaries and background for this paper. Section III illustrates our proposed *IFV* framework. The experimental results, including case studies and analysis of the effectiveness of the *IFV* framework, are presented in Section IV. Finally, the paper is concluded in Section V.

II. Preliminaries

A. Threat Model

For the threat model, we assume that attackers possess comprehensive knowledge of the system's functional behavior at the black-box level and have access to its input/output (I/O) interfaces. Through these interfaces, the attackers can analyze responses to extract security-critical information such as cryptographic keys. Furthermore, attackers are assumed to be capable of performing fault-injection attacks, such as clock or voltage glitching, as well as optical or laser-based attacks, to induce faults and gain access to internal design registers. Previous research has demonstrated correlations between fault-induced behavior of the design and information leakage [2], [27], [28]. Finally, we assume the presence of standard debug and test infrastructures, including scan chains and JTAG interfaces, which may inadvertently expand the attack surface. The primary focus of this paper is on the propagation and storage of sensitive data within an IP core. In a typical IP core, the protection mechanisms are less robust compared to dedicated secure memory or key management modules. To systematically assess such vulnerabilities, our objective is to develop a mathematical metric that enables the quantitative evaluation of information leakage and the estimation of the associated risk level across the datapath of an SoC design with high confidence.

B. Distance Metric

Spatial vulnerability plays a crucial role in assessing information leakage risks within hardware designs, particularly in terms of how sensitive information propagates among sequential cells (such as flip-flops and latches) and I/O ports. The *Distance Metric* is defined as the number of stages of sequential cells through which the sensitive data flows to an observable untrusted output port. This metric can be used to determine the quantitative security risk level associated with information leakage [9]. Sequential cells that are closer to the I/O ports exhibit a higher level of vulnerability, since the shorter distances through the logic gates provide fewer opportunities for sensitive data transformation or obfuscation before the data reaches an observable untrusted point. Previous attacks have demonstrated this correlation, showing that a small sequential depth significantly increases the likelihood of intercepting sensitive data at or near I/O buffers [20]. Moreover, hardware backdoors, whether unintentionally introduced through test and debug infrastructures or intentionally embedded within third-party IP (3PIP), pose an even greater threat when security-critical registers reside very close to such access points. As presented in [7], the spatial closeness between sensitive storage elements and debug or unauthorized access mechanisms amplifies the feasibility and impact of information leakage and data interception. Finally, the data propagation paths must be formally verified to eliminate the non-functional structural paths. In this manner, a high level of confidence can be obtained for the reported metric values.

III. IFV Framework

The high-level overview of the proposed *IFV* framework is illustrated in Fig. 1. There are three primary inputs to the framework. The first input is the RTL source code of a design. RTL source code for the design can be written in hardware description languages (HDLs) such as *Verilog*, *SystemVerilog*, or *VHDL*. Moreover, the RTL design may contain source code written in multiple languages. For instance, a portion of the RTL design can be written using *Verilog*, and the other part might be described utilizing *VHDL*. This type of RTL design is referred to as a *Mixed HDL* design. Hence, the proposed *IFV* framework is language-agnostic and independent of the complex syntax and constructs of HDLs. The second input to the framework is a technology library (a collection of standard cells). The technology library must be provided in *Library Database (.db)* and *Verilog (.v)* representations. The *Verilog* representation of the technology library contains the HDL description of the standard cells. The proposed *IFV* framework supports both open-source and proprietary standard cell libraries, making it technology-independent.

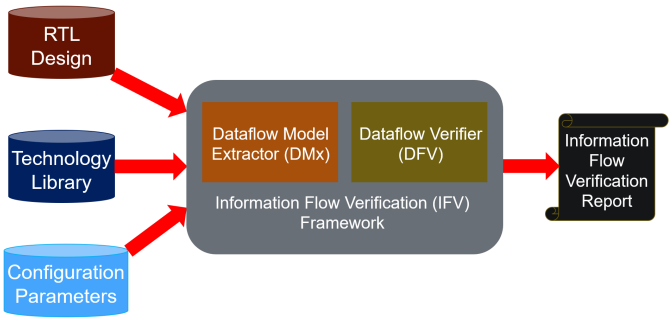


Fig. 1: A high-level overview of the *IFV* framework.

The third and final input to the *IFV* framework is a collection of configuration parameters for the framework. It needs to be provided in the *JavaScript Object Notation (JSON)* format. Such configuration parameters typically include the source and destination points of a path. These source and destination points might be a wire (single-bit wide net) or a bus (a collection of wires). Hence, the proposed *IFV* framework

can verify information flows at the bit level with fine-grained granularity. Moreover, the mode of analysis (can be either RTL or netlist), name of the top-level design module, file type of the top-level design module (can be *Verilog*, *SystemVerilog*, or *VHDL*), search path for the RTL source code of the design, compilation modes for the *Verilog* and *VHDL* languages, search path for the target technology library, and the technology library file in *.db* and *.v* formats. Additionally, the *IFV* framework supports black-boxing of design modules. This powerful abstraction technique significantly reduces the complexity of analysis. The design modules that need to be black-boxed are also included in the configuration parameters. Furthermore, flags can be set to perform design flattening and detect information leakage arising from the flow of sensitive data to the design's test and debug infrastructures, which are inserted in the design after performing logic synthesis. Thus, the proposed *IFV* framework can perform security assessment at both the RTL and gate-level netlist abstractions, making it a promising framework for information flow verification at the pre-silicon stage of the design lifecycle.

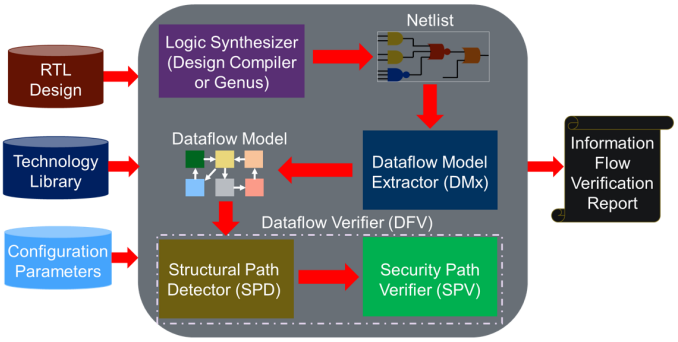


Fig. 2: A detailed overview of the *IFV* framework.

As shown in Fig. 1, the *IFV* framework comprises two primary modules: *Dataflow Model Extractor (DMx)* and *Dataflow Verifier (DFV)*. The *Dataflow Model Extractor (DMx)* module extracts the potential information flow model of the design. This model represents the structural connections among all flip-flops existing in the design. It constructs the connectivity graph of the flip-flops present in the design. By nature, this connectivity graph is directed and acyclic, representing the potential dataflow of the entire design. After obtaining the potential information flow model between the flip-flops in the design, the *Dataflow Verifier (DFV)* module performs the structural analysis of the flip-flop connectivity graph to detect potential information flow paths between the source and destination points. This structural analysis phase is purely static and therefore does not require generating any test vectors to identify the information flow paths between the source and destination points. After performing structural analysis, formal verification is conducted to determine the real information flow paths. The inclusion of formal verification has aided the *IFV* framework to prune non-functional structural paths. Furthermore, it has significantly reduced the amount of false positives and noise, which are the major drawbacks of any static verification methodology. A more detailed overview of the framework's implementation is illustrated in Fig. 2. After finishing the analysis, the *IFV* framework generates a comprehensive information flow verification report for the designer to review. The working principles of the building blocks of the proposed *IFV* framework are presented as follows.

A. *Dataflow Model Extractor (DMx)*

As illustrated in Fig. 2, the RTL source code of a design is first synthesized using a commercial logic synthesis tool. *Design Compiler* from *Synopsys* or *Genus* from *Cadence* can be utilized for this purpose. As mentioned before, two analysis modes can be set for the *IFV* framework. In RTL analysis mode, the RTL source code of the design is elaborated utilizing the logic synthesis tool. After performing RTL elaboration, a technology-independent netlist of the design is obtained. This technology-independent netlist comprises cells from the *Generic Technology (GTECH)* library, such as *GTECH_XOR, GTECH_OR, GTECH_AND, GTECH_INV, GTECH_DFF*, etc. In the netlist mode, the RTL design is synthesized, mapped, and optimized using the provided technology library. After this phase, the optimized gate-level netlist of the RTL design is obtained. Some configuration parameters (such as flags for performing design flattening or inserting DfT infrastructures) determine the type of the obtained gate-level netlist. For example, if the flag for design flattening is not set, a hierarchical gate-level netlist is generated. Otherwise, a flattened gate-level netlist is obtained. Moreover, suppose the flag for inserting the DfT and DfD infrastructures is enabled. In that case, a scan-chain inserted gate-level netlist is obtained. The elaborated or optimized gate-level netlist serves as the input for the *Dataflow Model Extractor (DMx)* module. An overview of the algorithmic flow of the *DMx* module is presented in Fig. 3.

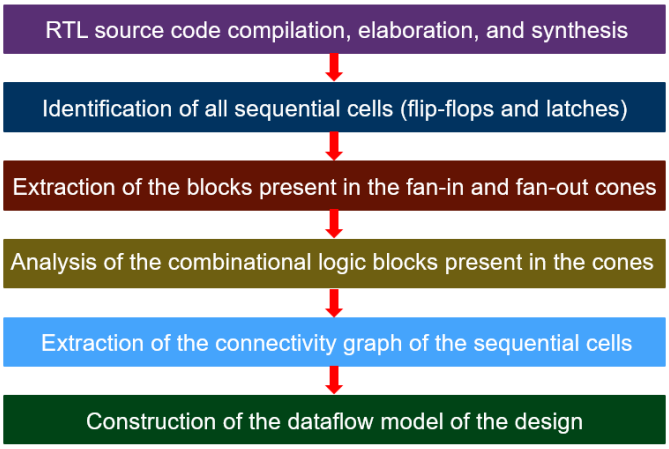


Fig. 3: Algorithmic flow of the *DMx* module.

First, the *DMx* module identifies all the sequential cells (flip-flops and latches) present in the design. Next, it extracts the combinational logic blocks present in the fan-in and fan-out cones of the detected sequential cells. Finally, after analyzing these logic blocks, the module obtains the connectivity information of the sequential cells. This connectivity information can be represented using the graph-based data structure. More precisely, it is a directed and acyclic graph. This graph is referred to as the *Dataflow Model (DM)*, as shown in Fig. 2. This model stands for the potential information flow between the sequential cells present in the design. Such a model serves as the foundation for the subsequent analysis phases.

TABLE I: Experimental results of *IFV* for open-source benchmarks collected from [11]–[13].

Benchmark	Source Point(s)	Destination Point(s)	Potential Path Count	Structural Path Count	Information Flow Path Count	Distance Metric
AES-T100	key[127:0]	Capacitance[63:0]	8,192	64	64	1
AES-T100-BB	key[127:0]	Capacitance[63:0], test_so	8,320	65	65	1
AES-T200	key[127:0]	Capacitance[63:0]	8,192	64	64	1
AES-T1700	key[127:0]	Antena[63:0]	8,192	64	64	1
AES-T2300	key[127:0]	Capacitance[63:0]	8,192	64	64	1
Pipelined AES	cipher_key[127:0]	cipher_text[127:0], valid_out	16,512	16,384	128	0
RISC-V Core	MEM_I_data[31:0]	O_DBG[63:0]	2,048	72	72	3 to 8
AES-LUT	key_in[127:0]	key_out[127:0]	16,384	16,384	0	N/A
CMAC Cipher	write_data[31:0]	read_data[31:0]	1,024	1,024	0	N/A
QT Processor	wbs_data_i[63:0], wbs_adr_i[63:0]	la_data_out[63:0]	8,192	871	869	1 and 3
DES Cipher	key[55:0]	desOut[63:0]	3,584	3,584	3,584	16
XTEA Cipher	write_data[31:0], cs, we	read_data[31:0]	1,088	64	48	0

B. Dataflow Verifier (DFV)

The *Dataflow Verifier (DFV)* module receives the generated dataflow model as its input. The algorithmic flow of the *DFV* module is illustrated in Fig. 4. Two major sub-modules act as the components of this module: *Structural Path Detector (SPD)* and *Security Path Verifier (SPV)*.

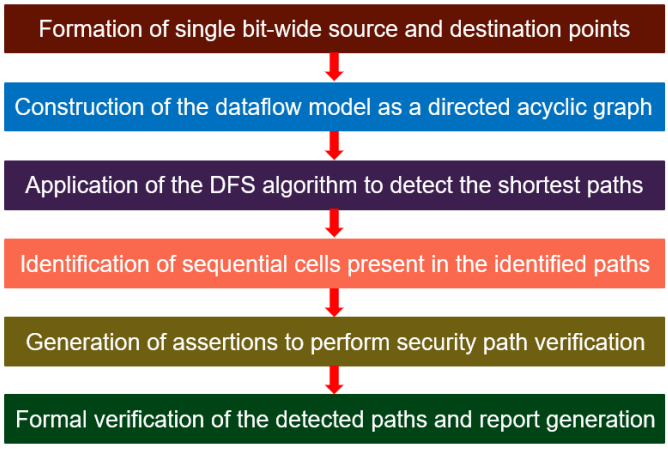


Fig. 4: Algorithmic flow of the *DFV* module.

The *Structural Path Detector (SPD)* sub-module obtains the dataflow model generated by the *DMx* module. Next, it performs scanning on the source and destination points of the paths obtained from the configuration parameter data. If the source and destination points are buses (multi-bit wide nets), they are transformed into single-bit wide nets or wires, as shown in Fig. 4. In this manner, the proposed *IFV* framework becomes capable of performing analysis at the bit level to identify information flow paths more precisely. Finally, the *SPD* sub-module performs graph algorithmic analysis to detect the structural paths existing between the source and destination points. It utilizes the *Depth-First Search (DFS)* algorithm to identify the shortest paths between the source and destination points, as presented in Fig. 4. The rationale behind identifying the shortest paths is that the shorter distances between the source and destination points reduce the likelihood of sensitive information being modified (mutated) and increase the possibility of data interception [9], [10]. It must be noted that these identified paths are, in fact, structural paths. Such paths might not always correspond to functional paths. To recognize the actual functional paths and filter out the non-functional structural paths, formal verification of these identified paths is essential, which can be achieved by performing security path verification using state-of-the-art commercial solutions.

The *Security Path Verifier (SPV)* sub-module is responsible for performing formal verification on the detected structural paths between the source and destination points. This sub-module receives the structural paths generated by the *SPD* sub-module as its input. After analyzing these paths, it automatically generates numerous formal assertions for performing security path verification. Such formal assertions are generated for each pair of sequential cells (flip-flops and latches) present in a particular structural path. These assertions are finally verified utilizing a formal security path verification tool. *JasperGold SPV* from *Cadence* or *VC Formal* from *Synopsys* can be utilized for this purpose. In this way, the information propagation paths through the flip-flops and latches are formally proven. During this process, non-functional structural paths are identified that may exist between the flip-flops and latches. Such structural paths are considered false paths and must be removed, since they do not represent real information flow paths. Moreover, the presence of such paths contributes to the generation of an enormous number of false positives and noise, adding a tremendous burden on designers to analyze and prune them manually. Due to the inclusion of formal security path verification, the proposed *IFV* framework is free from the issues arising from false positives and noise, which are inherent in any static (or structural) verification methodology. In this analysis phase, the sequential cells (flip-flops or latches) present in the information flow paths are extracted. Furthermore, the total number of flip-flops and latches is recorded, which has been defined as the *Distance Metric*, a notable security metric [9]. Both of these results are reported to the designer in a comprehensive information flow verification report. This report provides information on the source and destination points of the information flow paths, along with the corresponding distance metric (which can also be termed as depth), the flip-flops present in the paths, and the information propagation paths between the sequential cells.

In summary, the proposed *IFV* framework formally proves the distance metric utilizing both structural (static) analysis and formal security path verification. This metric can be used to calculate the security risk level for the security-critical assets quantitatively, as proposed in [9]. The inclusion of formal verification significantly enhances the designers' confidence in the reported distance metric. Such confidence cannot be achieved using the existing solution [9], which relies solely on structural analysis and is prone to generating a sheer volume of false positives and noise. Additionally, the proposed *IFV* framework enables bit-level analysis through the sequential cells present in a design, providing a comprehensive and precise information flow tracking of sensitive data at the bit level, which is unfortunately absent in the state-of-the-art commercial solutions, such as *JasperGold* or *VC Formal*. Finally, the *IFV* framework is technology library-agnostic. It

can be easily adopted at both the pre-synthesis (RTL) and post-synthesis stages of the design lifecycle. It makes the framework promising to the SoC design engineers.

IV. Experimental Results

A. Experimental Setup

The *IFV* framework has been implemented using the *Python* programming language. All experiments on the open-source designs were conducted using an Intel Core i7-1065G7 processor, with a 1.3 GHz clock frequency and 16GB RAM on a personal desktop. The RTL source codes of the designs were compiled, elaborated, synthesized, and optimized utilizing the *Design Compiler* tool from *Synopsys*. All the errors arising from various logic synthesis issues were identified and resolved correctly. Experimental results on 12 open-source RTL designs collected from numerous resources [11]–[13] are presented in Table I. In addition, the worst-case CPU time of the proposed *IFV* framework is illustrated in Table II. The total counts of gates for the benchmarks, as shown in Table II, were calculated from the report generated by *Design Compiler* after performing logic synthesis and optimization. The typical version of the *SAED90nm* library was used as the target technology library to synthesize the RTL designs. Nevertheless, the proposed *IFV* framework also supports other open-source and proprietary technology (standard cell) libraries.

B. Illustration of Experimental Results

As shown in Table I, the total number of potential paths was obtained by multiplying the bit widths of the source and destination points. The number of structural paths was achieved after performing graph algorithmic analysis on the source and destination points, which represents the total number of potential information flow paths. Finally, after performing security path verification using the *JasperGold SPV* tool from *Cadence*, the total number of information flow paths was obtained, representing the functional data propagation paths. It is evident from Table I that an enormous number of non-functional false paths can be present in the design. For instance, for the *AES-LUT* and *CMAC Cipher* benchmarks, the number of structural paths was 16,384 and 1,024, respectively. However, all of the structural paths were formally proven to be non-functional. Hence, the distance metric cannot be obtained for these benchmarks, since there exists no real information flow. For the other benchmark designs, the distance metrics were found and are presented in Table I.

Since the state-of-the-art solution proposed in [9] is entirely dependent on structural analysis to recognize information flow paths by constructing an asset flow graph (AFG), it cannot identify the actual information flow paths. Therefore, the distance metric results obtained from the proposed framework in [9] would be invalid. Conversely, the proposed *IFV* framework is free from such severe shortcomings, and it remarkably enhances the designer's confidence in the obtained results. It should be noted that a distance metric of 0 represents the absence of flip-flops or latches in the information flow path. In such scenarios, information from the source points propagates to the destination points via combinational logic blocks only. It is also apparent from Table I that the information flow paths may have multiple distance metric values.

In Table II, the worst-case CPU time of the *IFV* framework is presented for each of the open-source RTL designs. It includes the required time for performing logic synthesis and optimization, static verification for identifying the structural paths between the source and destination points, and formal security path verification for the detected structural paths. These results demonstrate that the proposed *IFV* framework can quickly and precisely identify information flow at the bit level in several open-source benchmark designs that vary widely in terms of functionality, size, and complexity. Additionally, case studies on two open-source benchmark designs are provided in the following sub-section to demonstrate the efficacy of the proposed *IFV* framework.

C. Case Studies

1) AES-T100: The *AES-T100* benchmark has been obtained from [11]. It contains the RTL source code of a cryptographic core (*AES*) that incorporates a *Hardware Trojan* within it. The inserted *Trojan* leaks the secret key through a hidden channel. A detailed description of it can be found in [11]. It resembles a *Always On Trojan*. The design is written using *Verilog* and was synthesized using the *SAED90nm* technology library using two methods. In the first method, the entire design was synthesized, resulting in a gate count of 285,531. On the other hand, in the second method, the module *aes_128* was black-boxed to simplify the analysis. For this scenario, the gate count of the design was 4,543, as shown in Table II. For the first case, the proposed *IFV* framework took 35 minutes to complete the entire analysis process. However, for the second scenario, it took only 12 seconds. In both cases, analysis was performed on the synthesized gate-level netlist of the *AES-T100* benchmark.

Source	Destination	Depth	Net(s)	Information Flow
key[0]	Capacitance[7]	1	\Trojan/N0	key[0] -> LOGIC_CLOUD -> \Trojan/N0 -> LOGIC_CLOUD -> Capacitance[7]
key[0]	Capacitance[6]	1	\Trojan/N0	key[0] -> LOGIC_CLOUD -> \Trojan/N0 -> LOGIC_CLOUD -> Capacitance[6]
key[0]	Capacitance[5]	1	\Trojan/N0	key[0] -> LOGIC_CLOUD -> \Trojan/N0 -> LOGIC_CLOUD -> Capacitance[5]
key[0]	Capacitance[4]	1	\Trojan/N0	key[0] -> LOGIC_CLOUD -> \Trojan/N0 -> LOGIC_CLOUD -> Capacitance[4]
key[0]	Capacitance[3]	1	\Trojan/N0	key[0] -> LOGIC_CLOUD -> \Trojan/N0 -> LOGIC_CLOUD -> Capacitance[3]
key[0]	Capacitance[2]	1	\Trojan/N0	key[0] -> LOGIC_CLOUD -> \Trojan/N0 -> LOGIC_CLOUD -> Capacitance[2]
key[0]	Capacitance[1]	1	\Trojan/N0	key[0] -> LOGIC_CLOUD -> \Trojan/N0 -> LOGIC_CLOUD -> Capacitance[1]
key[0]	Capacitance[0]	1	\Trojan/N0	key[0] -> LOGIC_CLOUD -> \Trojan/N0 -> LOGIC_CLOUD -> Capacitance[0]
key[1]	Capacitance[15]	1	\Trojan/N1	key[1] -> LOGIC_CLOUD -> \Trojan/N1 -> LOGIC_CLOUD -> Capacitance[15]
key[1]	Capacitance[14]	1	\Trojan/N1	key[1] -> LOGIC_CLOUD -> \Trojan/N1 -> LOGIC_CLOUD -> Capacitance[14]
key[1]	Capacitance[13]	1	\Trojan/N1	key[1] -> LOGIC_CLOUD -> \Trojan/N1 -> LOGIC_CLOUD -> Capacitance[13]
key[1]	Capacitance[12]	1	\Trojan/N1	key[1] -> LOGIC_CLOUD -> \Trojan/N1 -> LOGIC_CLOUD -> Capacitance[12]
key[1]	Capacitance[11]	1	\Trojan/N1	key[1] -> LOGIC_CLOUD -> \Trojan/N1 -> LOGIC_CLOUD -> Capacitance[11]
key[1]	Capacitance[10]	1	\Trojan/N1	key[1] -> LOGIC_CLOUD -> \Trojan/N1 -> LOGIC_CLOUD -> Capacitance[10]
key[1]	Capacitance[9]	1	\Trojan/N1	key[1] -> LOGIC_CLOUD -> \Trojan/N1 -> LOGIC_CLOUD -> Capacitance[9]
key[1]	Capacitance[8]	1	\Trojan/N1	key[1] -> LOGIC_CLOUD -> \Trojan/N1 -> LOGIC_CLOUD -> Capacitance[8]
key[2]	Capacitance[23]	1	\Trojan/N2	key[2] -> LOGIC_CLOUD -> \Trojan/N2 -> LOGIC_CLOUD -> Capacitance[23]
key[2]	Capacitance[22]	1	\Trojan/N2	key[2] -> LOGIC_CLOUD -> \Trojan/N2 -> LOGIC_CLOUD -> Capacitance[22]
key[2]	Capacitance[21]	1	\Trojan/N2	key[2] -> LOGIC_CLOUD -> \Trojan/N2 -> LOGIC_CLOUD -> Capacitance[21]
key[2]	Capacitance[20]	1	\Trojan/N2	key[2] -> LOGIC_CLOUD -> \Trojan/N2 -> LOGIC_CLOUD -> Capacitance[20]
key[2]	Capacitance[19]	1	\Trojan/N2	key[2] -> LOGIC_CLOUD -> \Trojan/N2 -> LOGIC_CLOUD -> Capacitance[19]
key[2]	Capacitance[18]	1	\Trojan/N2	key[2] -> LOGIC_CLOUD -> \Trojan/N2 -> LOGIC_CLOUD -> Capacitance[18]
key[2]	Capacitance[17]	1	\Trojan/N2	key[2] -> LOGIC_CLOUD -> \Trojan/N2 -> LOGIC_CLOUD -> Capacitance[17]
key[2]	Capacitance[16]	1	\Trojan/N2	key[2] -> LOGIC_CLOUD -> \Trojan/N2 -> LOGIC_CLOUD -> Capacitance[16]
key[3]	Capacitance[31]	1	\Trojan/N3	key[3] -> LOGIC_CLOUD -> \Trojan/N3 -> LOGIC_CLOUD -> Capacitance[31]
key[3]	Capacitance[30]	1	\Trojan/N3	key[3] -> LOGIC_CLOUD -> \Trojan/N3 -> LOGIC_CLOUD -> Capacitance[30]
key[3]	Capacitance[29]	1	\Trojan/N3	key[3] -> LOGIC_CLOUD -> \Trojan/N3 -> LOGIC_CLOUD -> Capacitance[29]
key[3]	Capacitance[28]	1	\Trojan/N3	key[3] -> LOGIC_CLOUD -> \Trojan/N3 -> LOGIC_CLOUD -> Capacitance[28]
key[3]	Capacitance[27]	1	\Trojan/N3	key[3] -> LOGIC_CLOUD -> \Trojan/N3 -> LOGIC_CLOUD -> Capacitance[27]
key[3]	Capacitance[26]	1	\Trojan/N3	key[3] -> LOGIC_CLOUD -> \Trojan/N3 -> LOGIC_CLOUD -> Capacitance[26]
key[3]	Capacitance[25]	1	\Trojan/N3	key[3] -> LOGIC_CLOUD -> \Trojan/N3 -> LOGIC_CLOUD -> Capacitance[25]
key[3]	Capacitance[24]	1	\Trojan/N3	key[3] -> LOGIC_CLOUD -> \Trojan/N3 -> LOGIC_CLOUD -> Capacitance[24]
key[4]	Capacitance[39]	1	\Trojan/N4	key[4] -> LOGIC_CLOUD -> \Trojan/N4 -> LOGIC_CLOUD -> Capacitance[39]

Fig. 5: Output report for the *AES-T100* benchmark.

As presented in Table I, the 128-bit secret key, *key[127:0]*, was considered as the source points. The leakage circuit, represented by a large capacitive load *Capacitance[63:0]*, was treated as the destination points. Additionally, for the second scenario, *DfT and DfD infrastructures* were inserted during the logic synthesis process utilizing *Design Compiler*. Hence, the observable debug port, *test_so*, was added to the list of destination points. For both cases, 64 information flow paths were detected. These paths originated due to the flow of the low-order byte of the secret key, *key[7:0]*, to the 64-bit wide capacitive load. It constitutes the *partial leakage* of the sensitive data, since only 8 bits out of 128 bits of the secret

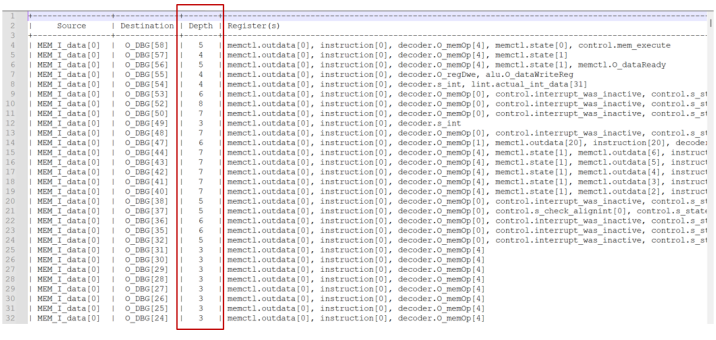
TABLE II: Worst-case CPU time of the proposed *IFV* framework for open-source RTL designs obtained from [11]–[13].

Benchmark Name	Gate Count	Logic Synthesis Time	Static Verification Time	Formal Verification Time	CPU Time
AES-T100	285,531	28 min.	5 min.	2 min.	35 min.
AES-T100-BB	4,543	5 sec.	5 sec.	2 sec.	12 sec.
AES-T200	285,548	25 min.	4 min.	1 min.	30 min.
AES-T1700	287,347	26 min.	4 min.	2 min.	32 min.
AES-T2300	287,927	27 min.	4 min.	2 min.	33 min.
Pipelined AES	247,134	36 min.	16 min.	20 min.	72 min.
RISC-V Core	82,158	4 min.	4 min.	2 min.	10 min.
AES-LUT	42,870	2 min.	1 min.	5 min.	8 min.
CMAC Cipher	39,266	12 min.	2 min.	1 min.	15 min.
QT Processor	36,064	2 min.	3 min.	8 min.	13 min.
DES Cipher	34,119	4 min.	1 min.	3 min.	8 min.
XTEA Cipher	5,690	10 sec.	7 sec.	4 sec.	21 sec.

key are actually leaked. Nevertheless, only one bit of the secret key, *key[7]*, leaks to the observable debug port, *test_so*.

Finally, for these two versions of the *AES-T100* benchmark, the distance metric was found to be 1, indicating a very close proximity between the sensitive information (the secret key) and the untrusted points (the leakage circuit and observable debug port). Eventually, it will lead to a high value for the security risk level. The information flow verification report generated by the tool implementing the proposed *IFV* framework is depicted in Fig. 5. As shown in the figure, the depth represents the obtained distance metric, and the nets stand for the data inputs of the flip-flops through which the secret key propagates to the untrusted points.

2) RISC-V Core: The open-source RISC-V processor core, written using *VHDL*, has been collected from [14]. It is a basic implementation of the RISC-V CPU. The implemented processor supports the *RV32IMZcsr* instruction set architecture (ISA). Moreover, the functionality of the core has been verified using an *FPGA* board [14]. The core has a 32-bit wide input port named *MEM_I_data[31:0]* for obtaining the binary version of a program (machine code) stored in the instruction memory (cache). This input port is security-critical and crucial to protect against information leakage, as it serves as the connection between the processor core and the instruction memory, which may store proprietary software programs or firmware. Additionally, the core features a 64-bit wide output port named *O_DBG*, which is used for debugging purposes, and is observable. This output port can be treated as an untrusted port, as an attacker can monitor it externally.



Source	Destination	Depth	Register(s)
MEM_I_data[0]	O_DBG[58]	5	memctl.outdata[0], instruction[0], decoder.O_memOp[4], memctl.state[0], control.mem_execute
MEM_I_data[0]	O_DBG[57]	4	memctl.outdata[0], instruction[0], decoder.O_memOp[4], memctl.state[1]
MEM_I_data[0]	O_DBG[56]	5	memctl.outdata[0], instruction[0], decoder.O_memOp[4], memctl.state[1], memctl.O_dataReady
MEM_I_data[0]	O_DBG[55]	4	memctl.outdata[0], instruction[0], decoder.O_regDwe, alu.O_dataWriteReg
MEM_I_data[0]	O_DBG[54]	4	memctl.outdata[0], instruction[0], decoder.s_int, lint.actual_int_data[31]
MEM_I_data[0]	O_DBG[53]	6	memctl.outdata[0], instruction[0], decoder.O_memOp[0], control.interrupt_was_inactive, control.s_st
MEM_I_data[0]	O_DBG[52]	8	memctl.outdata[0], instruction[0], decoder.O_memOp[0], control.interrupt_was_inactive, control.s_st
MEM_I_data[0]	O_DBG[50]	7	memctl.outdata[0], instruction[0], decoder.O_memOp[0], control.interrupt_was_inactive, control.s_st
MEM_I_data[0]	O_DBG[49]	3	memctl.outdata[0], instruction[0], decoder.s_int
MEM_I_data[0]	O_DBG[48]	7	memctl.outdata[0], instruction[0], decoder.O_memOp[0], control.interrupt_was_inactive, control.s_st
MEM_I_data[0]	O_DBG[47]	6	memctl.outdata[0], instruction[0], decoder.O_memOp[1], memctl.outdata[20], instruction[20], decode
MEM_I_data[0]	O_DBG[44]	7	memctl.outdata[0], instruction[0], decoder.O_memOp[4], memctl.state[1], memctl.outdata[4], instruct
MEM_I_data[0]	O_DBG[43]	7	memctl.outdata[0], instruction[0], decoder.O_memOp[4], memctl.state[1], memctl.outdata[5], instruct
MEM_I_data[0]	O_DBG[42]	7	memctl.outdata[0], instruction[0], decoder.O_memOp[4], memctl.state[1], memctl.outdata[4], instruct
MEM_I_data[0]	O_DBG[41]	7	memctl.outdata[0], instruction[0], decoder.O_memOp[4], memctl.state[1], memctl.outdata[3], instruct
MEM_I_data[0]	O_DBG[40]	7	memctl.outdata[0], instruction[0], decoder.O_memOp[4], memctl.state[1], memctl.outdata[2], instruct
MEM_I_data[0]	O_DBG[38]	5	memctl.outdata[0], instruction[0], decoder.O_memOp[0], control.interrupt_was_inactive, control.s_st
MEM_I_data[0]	O_DBG[37]	5	memctl.outdata[0], instruction[0], decoder.O_memOp[0], control.s_check_alignint[0], control.s_stat
MEM_I_data[0]	O_DBG[36]	6	memctl.outdata[0], instruction[0], decoder.O_memOp[0], control.interrupt_was_inactive, control.s_st
MEM_I_data[0]	O_DBG[35]	6	memctl.outdata[0], instruction[0], decoder.O_memOp[0], control.interrupt_was_inactive, control.s_st
MEM_I_data[0]	O_DBG[32]	5	memctl.outdata[0], instruction[0], decoder.O_memOp[0], control.interrupt_was_inactive, control.s_st
MEM_I_data[0]	O_DBG[31]	3	memctl.outdata[0], instruction[0], decoder.O_memOp[4]
MEM_I_data[0]	O_DBG[30]	3	memctl.outdata[0], instruction[0], decoder.O_memOp[4]
MEM_I_data[0]	O_DBG[29]	3	memctl.outdata[0], instruction[0], decoder.O_memOp[4]
MEM_I_data[0]	O_DBG[28]	3	memctl.outdata[0], instruction[0], decoder.O_memOp[4]
MEM_I_data[0]	O_DBG[27]	3	memctl.outdata[0], instruction[0], decoder.O_memOp[4]
MEM_I_data[0]	O_DBG[26]	3	memctl.outdata[0], instruction[0], decoder.O_memOp[4]
MEM_I_data[0]	O_DBG[25]	3	memctl.outdata[0], instruction[0], decoder.O_memOp[4]
MEM_I_data[0]	O_DBG[24]	3	memctl.outdata[0], instruction[0], decoder.O_memOp[4]

Fig. 6: Output report for the *RISC-V* processor core.

As presented in Table I, sensitive data (proprietary software programs or firmware stored in the instruction memory) can flow to the untrusted port (debug port). 72 structural flow paths have been detected out of 2,048 potential paths, and all of them are information flow paths that have been formally verified. For this *RISC-V* processor core, we obtained a set of distance metrics, ranging from 3 to 8. Therefore, the information flow paths have different levels of security risk, as noted in [9]. The information flow paths with a distance metric of 3 will have the highest security risk level. On the contrary, the paths with a distance metric of 8 will have the lowest security risk level. The generated report from the tool is shown in Fig. 6. The associated distance metric values have been illustrated.

The information flow paths with a low distance metric have severe security implications [15]–[17]. For these paths, attackers can easily observe the sequence of instructions stored in the memory. They can reverse-engineer the proprietary software program and algorithms, potentially leading to a significant loss of valuable intellectual property [15]. Moreover, they can alter the firmware or inject malicious code into the instruction memory. Such a scenario will allow them to modify the system behavior or bypass authentication mechanisms. Additionally, the attackers may manipulate the execution flow and security-critical parameters, which can help them to circumvent the intended program flow and security checking [16]. Furthermore, such information flow might lead to privilege escalation issues, as attackers may gain access to the software program stored in memory from a lower-privileged state [17]. These two case studies and the experimental results presented in Table I and Table II demonstrate the effectiveness of the proposed *IFV* framework in determining the distance metric more accurately with higher confidence compared to the existing solution proposed in [9]. In addition, the bit-level analysis capability of the *IFV* framework makes it more promising in terms of detecting information flow with greater precision and finer granularity than commercial solutions, such as *JasperGold* or *VC Formal*, which perform analysis at the word level for the sequential cells present in a particular data propagation path.

V. Conclusion and Future Work

The growing complexity of modern SoCs has significantly increased their vulnerability to information leakage and data integrity issues. In this paper, we have presented an information flow verification framework that integrates static and formal verification methodologies to identify covert data propagation paths. The proposed framework incorporates a novel mathematical metric for quantitatively assessing the security risks of detected paths, enabling designers to prioritize and

mitigate vulnerabilities arising from insecure data propagation early in the design lifecycle. By adopting formal verification, the proposed framework outstandingly enhances the designers' confidence in the reported metric values. Experimental results on 12 open-source benchmarks demonstrate the framework's ability to detect severe security issues at the pre-silicon stage effectively. We will focus on enhancing the proposed framework with automated mitigation techniques in the future. Moreover, we envision improvements in performance and scalability for industry-grade SoC designs.

REFERENCES

[1] W. Hu, C. -H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li, "An Overview of Hardware Security and Trust: Threats, Countermeasures, and Design Tools," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 6, pp. 1010-1038, 2021.

[2] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, and M. Tehranipoor, "AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs," 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 2016, pp. 1-6.

[3] N. Farzana, F. Rahman, M. Tehranipoor and F. Farahmandi, "SoC Security Verification using Property Checking," 2019 IEEE International Test Conference (ITC), Washington, DC, USA, 2019, pp. 1-10.

[4] M. Tehranipoor and F. Koushanfar, "A Survey of Hardware Trojan Taxonomy and Detection," in IEEE Design & Test of Computers, vol. 27, no. 1, pp. 10-25, Jan.-Feb. 2010.

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

[6] S. Akter, K. Khalil, and M. Bayoumi, "A Survey on Hardware Security: Current Trends and Challenges," in IEEE Access, vol. 11, pp. 77543-77565, 2023.

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

[8] R. Kibria, F. Farahmandi, and M. Tehranipoor, "ARC-FSM-G: Automatic Security Rule Checking for Finite State Machine at the Netlist Abstraction," 2023 IEEE International Test Conference (ITC), Anaheim, CA, USA, 2023, pp. 320-329.

[9] A. Ayalasomayajula, H. Li, H. Al Shaikh, S. K. Saha, and F. Farahmandi, "TDM: Time and Distance Metric for Quantifying Information Leakage Vulnerabilities in SoCs," 2024 IEEE 42nd International Conference on Computer Design (ICCD), Milan, Italy, 2024, pp. 130-133.

[10] M. Otto, "Fault attacks and countermeasures," Ph.D. dissertation, Paderborn University, Paderborn, Germany, 2004. [Online]. Available: http://www.martin-otto.de/publications/docs/DissertationMartinOtto.pdf

[11] Trust-Hub, https://trust-hub.org//benchmarks/chip-level-trojan

[12] FreeCores, https://github.com/freecores

[13] OpenCores, https://opencores.org/

[14] RISCV Core, https://github.com/Domipheus/RPU/tree/master/vhdl

[15] O. Shwartz, Y. Mathov, M. Bohadana, Y. Elovici, and Y. Oren, "Reverse Engineering IoT Devices: Effective Techniques and Methods," in IEEE Internet of Things Journal, vol. 5, no. 6, pp. 4965-4976, Dec. 2018.

[16] https://en.eeworld.com.cn/news/qcdz/eic701459.html

[17] Z. Ning and F. Zhang, "Understanding the Security of ARM Debugging Features," 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019, pp. 602-619.

[18] R. Kibria, F. Farahmandi, and M. Tehranipoor, "A Survey on SoC Security Verification Methods at the Pre-silicon Stage," Cryptology ePrint Archive, 2024.

[19] D. M. Anderson, "Design for manufacturability: how to use concurrent engineering to rapidly develop low-cost, high-quality products for lean production," CRC Press, 2020.

[20] R. Anderson, "Security Engineering: A Guide to Building Dependable Distributed Systems," Wiley, 2020.

[21] S. K. Saha, J. M. Mbongue, and C. Bobda, "Metrics for Assessing Security of System-on-Chip," 2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 2022, pp. 113-116.

[22] L. M. Reimann, L. Hanel, D. Sisejkovic, F. Merchant, and R. Leupers, "QFlow: Quantitative Information Flow for Security-Aware Hardware Design in Verilog," 2021 IEEE 39th International Conference on Computer Design (ICCD), Storrs, CT, USA, 2021, pp. 603-607.

[23] X. Guo, R. G. Dutta, J. He, M. M. Tehranipoor, and Y. Jin, "QIF-Verilog: Quantitative Information-Flow based Hardware Description Languages for Pre-Silicon Security Assessment," 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 2019, pp. 91-100.

[24] A. Meza and R. Kastner, "Information flow coverage metrics for hardware security verification," arXiv, 2023.

[25] J. Feldtkeller, T. G¨uneysu, and P. Schaumont, "Quantitative fault injection analysis," in Advances in Cryptology – ASIACRYPT 2023, J. Guo and R. Steinfeld, Eds. Singapore: Springer Nature Singapore, 2023, pp. 302–336.

[26] R. Purdy, D. Duvalsaint, and R. D. S. Blanton, "Security Metrics for Logic Circuits," 2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 2022, pp. 53-56.

[27] I. O. Somoye, J. Plusquellic, T. J. Mannos, and B. Dziki, "An engineered minimal-set stimulus for periodic information leakage fault detection on a RISC-V microprocessor," Cryptography, vol. 8, no. 2, 2024.

[28] D. Karaklajić, J. -M. Schmidt, and I. Verbauwhede, "Hardware Designer's Guide to Fault Attacks," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 21, no. 12, pp. 2295-2306, Dec. 2013.