QIF-Verilog: Quantitative Information-Flow based Hardware Description Languages for Pre-Silicon Security Assessment

Xiaolong Guo[*], Raj Gautam Dutta[*], Jiaji He[*†], Mark M. Tehranipoor[*], and Yier Jin[*]
[*]Department of Electrical and Computer Engineering, University of Florida
[†]School of Microelectronics, Tianjin University
guoxiaolong@ufl.edu, r.dutta@ufl.edu, dochejj@tju.edu.cn, tehranipoor@ece.ufl.edu, yier.jin@ece.ufl.edu

***Abstract*—Hardware vulnerabilities are often due to design mistakes because the designer does not sufficiently consider potential security vulnerabilities at the design stage. As a result, various security solutions have been developed to protect ICs, among which the language-based hardware security verification serves as a promising solution. The verification process will be performed while compiling the HDL of the design. However, similar to other formal verification methods, the language-based approach also suffers from scalability issue. Furthermore, existing solutions either lead to hardware overhead or are not designed for vulnerable or malicious logic detection. To alleviate these challenges, we propose a new language based framework, QIF-Verilog, to evaluate the trustworthiness of a hardware system at register transfer level (RTL). This framework introduces a quantified information flow (QIF) model and extends Verilog type systems to provide more expressiveness in presenting security rules; QIF is capable of checking the security rules given by the hardware designer. Secrets are labeled by the new type and then parsed to data flow, to which a QIF model will be applied. To demonstrate our approach, we design a compiler for QIF-Verilog and perform vulnerability analysis on benchmarks from Trust-Hub and OpenCore. We show that Trojans or design faults that leak information from circuit outputs can be detected automatically, and that our method evaluates the security of the design correctly.**

I. Introduction

The changing landscape of the semiconductor industry has increased the demand for intellectual property (IP) cores. Various factors, such as shortened time to market (TTM) and lowered design cost, have led to the proliferation of the IP market. Meanwhile, the impact of malicious logic and design flaws in IP cores threatens to ruin the credibility of third-party vendors and places unnecessary security risks on the IP customers and end users. Existence of a malicious IP core in a system-on-chip (SoC) invalidates the applicability of many of the previously proposed methods for hardware Trojan detection [1], [2].

Most of the hardware vulnerabilities are the result of designers not addressing security problems adequately [3]. With growing complexity of SoC designs, the workload is overwhelming for SoC designers to manually diagnose security vulnerabilities. In addition, mitigating vulnerabilities after the design stage results in increased costs and delayed TTM. Therefore, developing automated methods for detecting and evaluating vulnerabilities in the design stage is highly desired.

Hardware vulnerabilities could result in information leakage. With sensitive secrets like user privacy, passwords, or encryption keys routinely being stored and communicated in our daily use of electronic devices, data secrecy protection has become a key objective of computer security research [4], [5]. Although cryptographic algorithms play an important role in preventing key leakage, adversaries still exert all their focus and effort to access sensitive information via weaknesses appeared in the hardware implementation.

Information flow tracking (IFT) [6] is a powerful approach for preventing sensitive information leakage. In this approach, labels representing secrecy/trust levels are assigned to data and operations on data are extended to include operations on their labels, based on predefined information flow policies. Access or propagation of data with sensitive labels is thereby restricted to trusted segments of the code or system and is forced to abide by the desired information flow policies. SecVerilog [7], [8], Caisson [9] and Sapper [10] are some of the solutions based on IFT. However, these solutions either require developers to have sufficient knowledge to insert the labels precisely, or they suffer from hardware overhead.

On the other hand, Quantitative Information-Flow (QIF) models and analyses provide such a technique for characterizing the magnitude of information leaks [11]–[13]. Specifically, for a classical QIF model, from observation of both program behaviors, uncertainty is represented with probability distributions. The QIF metric is then calculated to quantify information leakage depending on those distributions. Mardziel et al. [13] put a dynamic QIF model and metric in a programming language, but there is no attempt of applying QIF to hardware with automatic and applicable tool.

In the proposed approach, we extend the type system of Verilog to provide only one new type for labeling signals. Developers can attach the new label to any signals (wires or registers) in designing hardware. We name the new language as QIF-Verilog which requires developers to only find out "secret" without considering any IFT policies. The QIF metrics named Remaining Uncertainty (RU) and Accumulated Remaining Uncertainty (AR) for quantifying hardware information leakage are proposed as a reference to terminate taint/labels propagation. In the end, we check whether the output/inout ports are tainted by the sensitive label. Also, developers can obtain a metric AR with the most vulnerable propagation path which represents the barrier of accessing labeled sensitive information by adversaries.

The main contributions of this paper are as follows:

- We propose a new hardware description language named QIF-Verilog, extended from Verilog, that tracks information flow of the specific type signals through using taint propagation. By using the QIF-Verilog, developers with no background on security can intuitively label the sensitive signal and quickly evaluate the design's trustworthiness.
- A QIF model is designed to apply information-theoretic metric for quantifying flow of sensitive secrets in hardware design. The model includes metrics that can quantify the sensitive downgrades of secrets in transition in the data flow. To the best of our knowledge, we are the first to design and apply QIF based model and metric for hardware-level information leakage prevention.
- We put the QIF model and metric into use through implementing them in the proposed new language. Then we design and evaluate a series of secure benchmarks to show that information leakage vulnerabilities can be detected in design stage. Further, our approach can provide security assessment to large systems, thereby addressing the scalability challenge.

The rest of the paper is organized as follows: In Section II, we discuss previous work on malicious logic detection using QIF-Verilog based solutions and present QIF model and metric, then discuss how our work differs from them. In Section III, we introduce the threat model and provide some relevant background on type system for IFT, and QIF for information leakage. We explain our extended language, assessment model, and RU metric in Section IV. The threshold setup and tool development are also introduced in this section. In Section V, the derivations of RU equations for several representative operations are discussed in detail with examples. Section VI presents demonstrations of our approach. Finally, conclusions and future works are drawn in Section VII.

II. Related Work

A. *Language-based Information Flow Security in Hardware*

Protecting the confidentiality of sensitive information is one of the fundamental security objectives of IFT methods. Traditional IFT enforces the noninterference policy, which states that low outputs are independent of high inputs; this implies that an adversary can deduce nothing about the high inputs from the low outputs. There are many existing works that can check the noninterference property by using type systems and Sabelfeld et al. [14] wrote a survey about these works.

Recently, two languages for hardware security have been developed, Caisson [9] and Sapper [10], which synthesize secure circuits that satisfy IFT isolation and separation properties. In both these languages, the designer needs to set up security labels for all signals, specifically wires and registers. All hardware items, like registers, relevant to information flow are duplicated in a program written by Caisson. Although improvement of Caisson is made by applying dynamic type system in Sapper, hardware overhead will still be caused at the circuit level.

SecVerilog extends type system of standard Verilog [8] and avoids the hardware overhead by performing verification in the compilation stage. To enforce noninterference, [8] designs a predication mechanism in SecVerilog, which implies that an attacker cannot infer information about the secrets. However, because lots of useful information depends on high sensitive secrets, achieving noninterference is generally not possible. For example, rejecting an incorrect password will exclude a wrong guess of the correct password. Thus, to increase the precision, SecVerilog leads to significant complexity of adding security labels. Applying SecVerilog, developers have to tag security labels very carefully to wires and registers in the Verilog design. In other words, circuit designers must consider many security rules to specify information flow policy while adding labels. Therefore, SecVerilog is difficult for developers that do not have an understanding of security features of the design. From a developer's perspective, a simplified (or automatic) solution is a preferred alternative.

Instead of enforcing noninterference, the quantitative theory of information flow aims at relaxing noninterference, which quantifies how much information is being leaked [11]. Through tolerating a possibility of leaking information, the quantitative information flow provides a reference to reduce the complexity of realizing information flow policy significantly. In the QIF-Verilog, as information flow policy is assembled in the proposed QIF model, only one new type is defined and used to highlight the secret, which is explained with an example in Listing 2. Comparison is also made with SecVerilog, Listing 1. In fact, in addition to the highlighted labels, there are other separate files further defining dependent types.

Listing 1 SecVerilog [15]

```
module deptype(...);
 input[15:0] {L} timer;
 input[15:0] {H} data;
 reg[1:0] {Par cur_state}
          cur_state;
 reg[1:0] {Par next_state}
          next_state;
      ...
```

Listing 2 QIF-Verilog

```
module deptype(...);
 input[15:0] timer;
 Taint input[15:0] data;
 reg[1:0] cur_state;

 reg[1:0] next_state;

       ...
```

B. *Quantified Information Flow*

Based on information theory, the key idea of QIF is to assess the quantity of the leaked sensitive information to insecure channels through information flow [11]. In [12], Kopf and Basin present a QIF model for adaptive attacks on deterministic systems based on the assumption of uniform prior distributions. Bounds of leaked information in the above system are deduced in the approach. [13] uses Kopf and Basin's work to measure secret information leakage with time change. Specifically, several metrics of uncertainty are considered including vulnerability [11], g-vulnerability [16], and Guessing-entropy [17]. Authors in these papers design a QIF model and put it into a functional language. However, as programmers rarely use functional programming to develop large-scale circuits, the language in [13] is not applicable for

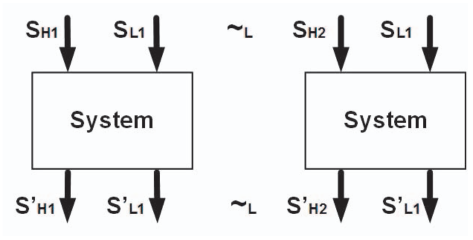

Figure 1: Noninterference model

current practices. Furthermore, there is no attempt to apply QIF in the hardware security area.

III. Background

A. Attack Model

In this paper, we assume that the untrusted hardware leaks information through the leakage path caused by design errors. Specifically, lack of security knowledge of the hardware designers could result in vulnerabilities such as leakage paths, in the design stage. The adversaries can take advantage of such vulnerabilities to leak information. We assume that the adversary has knowledge of the functionality of the design and have physical access to I/O ports of the manufactured hardware. By providing inputs and observing outputs, the adversary derives sensitive information such as encryption keys from the hardware.

B. Noninterference and Information Flow Labels

Confidentiality of a system is presented as, while the inner structure of the system is treated as a black box, an adversary learns no more information from system executions than from direct observations. We define a state of system as S, and then the state S is divided into low part and high part - such as S_H and S_L. The low part states S_L is in public domain, which can be observed by an adversary, while the high part states S_H is the one we do not want an adversary to learn. Here, inner structure of the system is treated as a black box.

When the system is executing, initial states $S_{H1} \in S_H$ and $S_{L1} \in S_L$ are its input. Consequently, the output states of the system are $S'_{H1} \in S_H$ and $S'_{L1} \in S_L$. When there is a second execution of the system, the low part states input-output, S_{L1} and S'_{L1} remain the same, whereas the high part states change value i.e., $S_{H2} \in S_H$ and $S'_{H2} \in S_H$ are new inputs and outputs of the system, respectively. A description of this procedure is given in the Figure 1.

In the above system, since adversaries cannot learn the difference between $\{S_{H1}, S'_{H1}\}$ and $\{S_{H2}, S'_{H2}\}$, we define the indistinguishable relation under observation of S_L states as $\sim_L$. Further, let $[[s]]$ be behaviors starting from states S, like execution starting from S. Then, the Noninterference given by [11] can be represented as follows,

$$S_1 \sim_L S_2 \Rightarrow [[S_1]] \approx_L [[S_2]] \tag{1}$$

where, $\approx_L$ defines similar indistinguishable relations in behaviors. Equation (1) means that starting from two states S_1 and S_2, if the initial states are indistinguishable, then their behaviors will also be indistinguishable. Actually, different confidentiality properties can be captured by properly choosing between $\sim_L$, $[[\bullet]]$, and $\approx_L$. For language-based IFT solutions, $[[\bullet]]$ stands for semantics of design or program, while the two relations $\sim_L$ and $\approx_L$ imply what an attacker can observe.

All previous language-based IFT methods set up sensitive levels from a partially ordered set and determine the security policy. A commonly followed practice is to design two levels H (high sensitive, Private) and L (low sensitive, Public), then restrict H labeled signals flowing to L labeled signals, while the reverse is allowed. In the proposed QIF-Verilog, we relax the noninterference by quantifying the threats of information leakage inner the system.

C. QIF Model and Metrics

Although maintaining noninterference is important for protecting confidentiality of a system, it is not practical in hardware design because S_L strongly depends on S_H in many scenarios. For instance, in all the encryption hardware, both plaintext and ciphertext belong to S_L, however, ciphertexts are obtained from interoperation among plaintext and key which belongs to S_H. Therefore, there are a variety of approaches to relax noninterference [18] and QIF provides a quantitative solution of accounting how much information is leaked.

In [11], a practical QIF model is summarized where the system produces S_L as output while receiving S_H as input. Attackers attempt to infer information from S_H by observing signal S_L. Under these assumptions, the amount of information leaked to the attacker can be expressed as,

Definition 1. *Information Leaked = Initial Uncertainty - Remaining Uncertainty*

From an attacker's perspective, information leakage is the change of uncertainty during observation. Generally, uncertainty is deduced using probability distributions, specifically, normal distribution in our approach. Then, QIF metrics are calculated in the form of a particular number, which stands for information leakage.

Accordingly, vulnerability $V(S_H)$ is defined as worst-case probability that attacker could deduce the value of S_H correctly in one try [11]. Other assumptions are that system is deterministic and S_H is uniformly distributed, and we define $|S|$ as the number of possible states in S. Then the min-entropy of S_H, $H_\infty(S_H)$, is defined as,

$$H_\infty(S_H) = log\frac{1}{V(S_H)} = log|S_H| \tag{2}$$

where, log is of base 2. If S implies states of a register, value of $log|S|$ is the length of the register. Correspondingly, conditional min-entropy $H_\infty(S_H|S_L)$ is defined as,

$$H_\infty(S_H|S_L) = log\frac{1}{V(S_H|S_L)} = log\frac{|S_H|}{|S_L|} \tag{3}$$

Thus, we quantify various uncertainties and information leakage as follows:

$$\textit{Initial Uncertainty} = H_\infty(S_H)$$

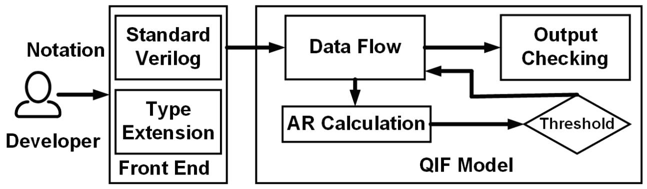

Figure 2: Working procedure of the proposed approach.

$$\textit{Remaining Uncertainty} = H_{\infty}(S_H|S_L)$$

$$\textit{Information Leaked} = H_{\infty}(S_H) - H_{\infty}(S_H|S_L)$$

Finally, information leaked is $log|S_L|$.

IV. Methodology

The proposed QIF-Verilog denotes sensitive information for behavioral Verilog and then statically checks information flow in a quantitative way. By adding a single type, our method can enable an information flow policy on secret propagation to relax noninterference in the design. Metrics are designed to quantify sensitive downgrading along the propagation from secrets to output ports. The entire design, including secrets with the secure label, are parsed in data flow and then the QIF model is applied to perform analysis by comparing the metrics with a threshold. Accordingly, we develop a compiler to realize the proposed QIF-Verilog.

A. A Highly-Secure Hardware Description Language

Extending Verilog types, we design a highly-secure HDL language by adding a new type, $Taint$. Compared to prior secure languages, the security lattice that maps to security levels only contains one single element. Hence, as shown in Equation (4), the syntax of security label in this paper is merged with Verilog signal types such as *wire* and *reg*.

$$sigtype : INPUT|OUTPUT|REG|WIRE|\ldots|Taint \quad (4)$$

where $sigtype$ stands for a set of all signal types.

The working procedure of our proposed approach is shown in Figure 2, where a developer designs circuit using the QIF-Verilog and assigns secrets with type $Taint$. Then, statically the designs with new notations are parsed to data flow graphic (DFG), which is processed by the QIF model. Applying the QIF model, the labels are propagated from tainted nodes, defined as transition source, to the untainted nodes, defined as transition destination, along with data flow. A pair of transition source and destination, together with corresponding threshold comparing, formed a transition in our QIF model. An example of a transition and the metrics are shown in the Figure 3. In IFT systems, information flow policies are defined to configure the taint source, taint target and tracking rules for taint propagation. These definitions of the QIF model are presented in the following sections. Specific tracking rules are designed to address the leakage vulnerabilities of the RTL design.

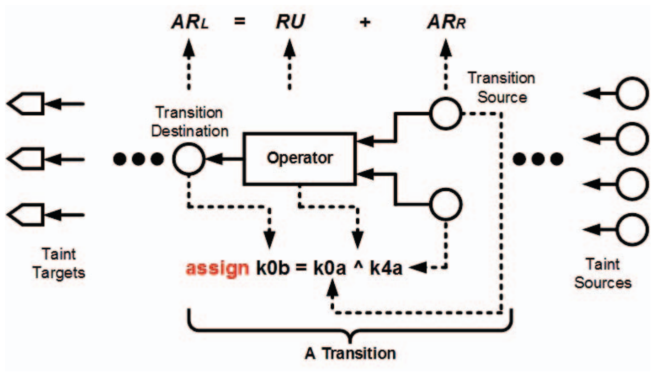

Figure 3: Taint propagation from the taint sources to the taint targets.

1) Taint Source: The signals declared in $Taint$ type are treated as storing sensitive information. As the entire design is represented with the help of a DFG, all the signals are mapped to the nodes of the DFG. The nodes converted from $Taint$ type signals are defined as the taint sources.

2) Taint Targets: The signals with types Out and $In-Out$ in the top module of a design are defined as the taint targets. The sensitivity of a signal will be downgraded during the propagation until it becomes insensitive. In the end, all the output ports (including in-out ports) will be checked to find out if they are sensitive. If so, an alarm will be triggered and the vulnerable propagation path will be displayed. Otherwise, the design will be treated as satisfying the confidentiality property. Then, Verilog code is generated by removing the extended types, $Taint$.

3) Taint Propagation: As mentioned in the introduction section, the taint propagation is composed by a series of transitions as shown in Figure 3. Specifically, the sensitive label $Taint$ is propagated from transition source to transition destination. During the operation between transition source and destination, a metric RU is calculated to quantify the sensitive downgrading in secrets propagation[1]. Then an Accumulated RU (AR), AR_L in Figure 3, on each transition destination is obtained from adding RU to AR, AR_R, on corresponding transition source. The AR on each taint source node is set to zero. The sensitivity in the propagation is downgraded with the increasing of an AR. If an AR reaches a threshold, the corresponding propagation is terminated, which means that the $Taint$ label will not be added on the transition destination node. The threshold is computed based on the length of secrets; a detailed explanation is given in the Sub-section IV-C.

The typing rules for propagation of security label $Taint$ are shown in Equations (6) and (7). Explicitly, there are two propagation rules: 1) via assignment, either blocking or non-blocking, as presented in Equation (6), and 2) via implicit statement as presented in Equation (7). In both rules, most of the notations are similar to those of [19].

[1]We assume there are very high sensitive levels for secrets. The sensitive levels will be propagated to other signals connected to secrets and changed during propagation.

Operator (op)	T op U	T op T
ASSIGN/NOT	0	
LSHIFT/RSHIFT	0	
DECONCATENATE	$W - W'$	
CONCATENATE	0	0
AND/NAND	$\frac{W}{2}$	W
OR	$\frac{W}{2}$	W
XOR/XNOR	$\frac{W}{2}$	W
PLUS/MINUS	$\frac{W}{2} - 1$	$W - 1$
MOD/DIVIDE	$W(\frac{W}{2} - 1)$	$W(W - 1)$
TIMES	$\frac{W}{2} + (W-1)^2$	$W + (W-1)^2$
POWER	$W + (W-1)^2$	$W + (W-1)^2$
LT/GT	$W - 1$	$2W - 1$
LE/GE	$W - 1$	$2W - 1$
EQ/NE	$W - 1$	$2W - 1$
COND	$W - 1$	$2W - 1$

Table I: RU of a single step transition in data flow.

For instance, in Equations (6) and (7), sig means signal in the system while Γ is the context of types and **FS(**$\Gamma, Taint$**)** returns all the signals that have been labeled $Taint$ in Γ. Symbol **exp**(sig) stands for single-step execution expression which includes sig. Function **AR**(sig) returns a QIF metric in the signal sig, which is then compared to a threshold Th. We show the details of **AR** in the following subsection. pc is defined as program-counter to trace control flows and $\mathscr{M}$ monitors all signals' changes by alternative executions. Analysis function **DA**(η) returns variables that have to be assigned by any execution at location η.

B. Vulnerability Quantifying Model and Metrics

As tainted sources, secrets have a very high sensitivity level in a design. Intuitively, in the information flow tracking, once a secret is involved in an operation, there is a possibility of leaking information. On the other hand, from adversaries' perspective, compared to getting access to the secret directly, barrier for guessing the secret will be increased with more transitions between secret and the observation point. In other words, during information propagation, the sensitivity of secret should tend to be downgraded to null (no sensitivity). In the following section, we show the details of building the quantifying model and designing the metric that can evaluate the vulnerability of secret propagation through information flow tracking.

From rules of label propagation in Equations (6) and (7), the function **AR** dominates whether the security label $Taint$ will be propagated from transition source signal sig_R to transition destination signal sig_L. In our approach, **AR** receives the transition destination node in data flow and returns an Accumulated Remaining Uncertainty value on this node as mentioned above. Specifically, as a single-step execution in QIF-Verilog maps to a transition in data flow, an RU, shown on Figure 3, should be calculated for every transition in propagation accordingly.

For every kind of operator in QIF-Verilog, we define the corresponding RU equation to compute RU. We make two assumptions to realize the QIF model and RU equation,

Assumption 1. *Binary values in all of the signals in the hardware system, including wires, registers, and memories are uniformly distributed.*

Assumption 2. *If two signals from the same source are separated far apart, we do not take their dependency into consideration in calculating entropy.*

For Assumption 1, considering that many encoding methods, like spread spectrum encoding, are used to let 1 and 0 equally appear in bottom circuit, it is reasonable to assume all the signals get high or low voltages in the same probability. The distribution is used when we evaluate the remaining uncertainty of each operation in the Verilog program. Thus, changing from uniform to other distributions will only require changes in Table I. We will improve the applicability of the QIF-Verilog by applying more appropriate types of distributions to different hardware components in future.

In Assumption 2, the rational is that the application of original QIF is narrowed to a single-step execution. In the following section, we set a rule to make a selection when two secrets propagation paths merge. The rule can help cancel the correlations in reconvergent fanout. Hence, even if two wires coming from the same node have a dependency, each wire will contribute RU to its own propagation, accordingly. We will discuss more details about the merge of propagation in Section V.

A specific RU is calculated on each single step transition based on the real hardware program. Then the RU is treated as an attribute value of the destination node.

$$sig_L \Leftarrow \text{exp}(sig_R) \tag{5}$$

As in Equation (5), corresponding RU generated in **exp**(sig_R) will be put into sig_L, which is defined as a destination signal in the assignment. Also, we let **RU**(sig_L) return the generated RU. Then, for each node in the information flow, an accumulated RU is computed by considering all previous relevant steps as shown in Equation (6).

$$\text{AR}(sig_L) = \text{RU}(sig_L) + \text{AR}(sig_R) \tag{6}$$

From Equation (3), by giving sig_{R1} and sig_{R2} as inputs, and then producing sig_L, RU generated from **exp**(sig_L) is $log\frac{|sig_{R1}|+|sig_{R2}|}{|sig_L|}$, explicitly $|sig_{R1}| + |sig_{R2}| - |sig_L|$ which is actually an equation about signal lengths. In the proposed method, we deduce the corresponding RU equation for each kind of operator and show the results in Table I. In the source code level, operators are involved in the transition, which are shown in the first column of the table. For the rest of the columns, RU equations are given by considering signal's label and length. Meanwhile, in the first row of the table, T stands for signals labelled by $Tainted$ while U stands for the rest untainted signals. op means operator applied, accordingly. We also assume that W is the length of signal involved in the transition.

$$\frac{\Gamma \vdash sig_R : Taint, sig_L \notin \text{FS}(\Gamma, Taint) \models P(\bullet\eta), \text{AR}(sig_L) < Th \Rightarrow \begin{matrix}\Gamma, pc, \mathscr{M} \vdash sig_L =_\eta \text{exp}(sig_R) \\ \Gamma, pc, \mathscr{M} \vdash sig_L \Leftarrow_\eta \text{exp}(sig_R)\end{matrix}}{\Gamma, pc, \mathscr{M} \vdash Taint \bigsqcup pc = Taint} \quad \text{T-PropagationA} \tag{6}$$

$$\frac{\Gamma \vdash \text{exp}(sig_R) : Taint, \text{AR}(sig_L) < Th \Rightarrow \Gamma, pc, \mathscr{M} \vdash \text{if}_\eta(\text{exp}(sig_R))c_1 \quad \text{else} \quad c_2}{\Gamma, Taint \bigsqcup pc, \mathscr{M} \cap \text{DA}(\eta) \vdash c_1 \quad \Gamma, Taint \bigsqcup pc, \mathscr{M} \cap \text{DA}(\eta) \vdash c_2} \quad \text{T-PropagationIF} \tag{7}$$

As an example, in Figure 3 the transition destination $k0b$, which maps to sig_L above, is assigned by an exclusive OR operation between the transition sources $k0a$, which maps to sig_{R1}, and $k4a$, which maps to sig_{R2}. We assume that $k0a$ includes the $Taint$ label, while $k4a$ does not. Then in this transition, AR_R, which maps to $AR(sig_R)$, is obtained from node $k0a$. In the exclusive OR operation, RU is produced depending on the row beginning with XOR and column T op U in Table I, which equals to half the value of the length of the signal $k0a$. And then we can calculate the Accumulate RU AR_L, which maps to $AR(sig_L)$ above, on node $k0b$ by adding RU and AR_L.

On the other hand, from the Equation (6), the accumulated RU in the transition destination is obtained by adding the computed RU in the transition destination to accumulate RU in transition source. When two secret propagations merge together (tainted signal operates with tainted signal), we define the following rule to make a selection between two RUs in transition sources - when both transition sources are tainted signals, considering that the propagation path containing less accumulated RU is more suspicious in leaking information, the tainted transition source signal carrying less accumulated RU will be preserved while the one carrying more AR will be discarded. The rules further result in the strategy of preventing the cyclic paths during the propagation. As the AR in parents nodes are less or equal to their children nodes in the DFG, we set the strategy as: canceling all the cycles between the nodes and their parents.

C. Threshold of Terminating Propagation

The threshold is automatically computed by considering the average length of initially labeled secrets. There are two concerns in computing the threshold - a lower bound must be higher than the accumulated RU in vulnerable propagation, while an upper bound must be lower than the accumulated RU in normal propagation. Heuristically, more gates involved in propagation will increase the barrier of guessing secret by attackers. From Table I, all the logic gates bring in linear RU increasing. Thus, we design the threshold as a linear relationship with an average length of secrets. To estimate the value of the parameter in the linear relationship, we hope there is a full diffusion, which means a significant sensitivity downgrade, between the secrets and observation points in any arbitrary design. We assume that for per unit length secret, to get the full diffusion, the AR value from different benchmarks are approximately equal. Therefore, the parameter can be deduced from a specific use-case, like following AES. Depending on [20], a full diffusion is obtained through 2 rounds inner AES. For each inner round of 128 bits AES, we compute the accumulated RU from input key to output key as 159 based on Table I. Then the parameter in the linear relation is $2 \times \frac{159}{128}$, which is 2.49. This means that in the case of the full diffusion, it is 2.49 Accumulated RU is required for one bit secret. Hence the threshold is:

$$Th = 2.49 \times avg(sig : Taint) \tag{7}$$

where avg returns the average value of the secrets' length. Note that the 2.49 computed here is a generally used parameter for arbitrary benchmarks.

D. Automatic Tool Development

Subsequently, a compiler written in Python is developed, which receives the QIF-Verilog program as input and on which IFT analysis is performed. Note that in implementation, the function of parsing QIF-Verilog codes as a data flow graph is developed by enhancing PyVerilog [21]. Each signal in the RTL design maps to a node in the DFG. The proposed metrics, such as RU and AR, are designed as parameters of the node. Connections among signals are identified as edges among nodes. An automatic taint tracking mechanism is developed to realize the information flow policy.

Specifically, in the QIF model, the metrics are used to analyze the data flow. All the AR in taint sources are set to be 0 at the beginning. From the tainted nodes, the tool searches the transition destinations and then propagates the taint label. The RU is calculated along with the taint propagation. Accordingly, accumulated remaining uncertainty (AR) is computed from RU in each transition depending on (6). Then a taint propagation would be determined once the AR on this propagation is larger than the threshold. If the tainted label is detected in any output ports, an alarm will be raised indicating the violation of the confidentiality property.

V. Remaining Uncertainty Derivations

As mentioned earlier, a metric RU in each transition is computed based on the equations provided in Table I. In the following section, we show the derivation details of the Remaining Uncertainty through several examples and discuss the propagation merges in details.

A. Operations - Single Input

We define the single input operation as there is only one node in transition source involved in the operation. The QIF model for single input operators, like ASSIGN, NOT, SHIFT, is given in Figure 4(a). As observed from transition destination, information from the transition source is leaked for ASSIGN. Thus, there is no uncertainty for adversaries and the RU produced from the ASSIGN operation should be 0. For

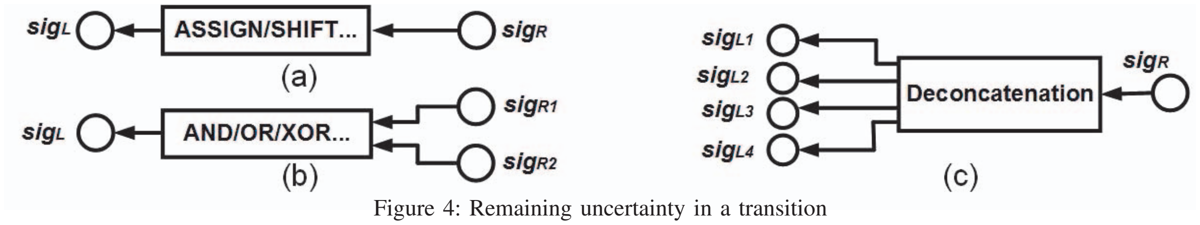

Figure 4: Remaining uncertainty in a transition

the operators SHIFT and NOT, although there are changes between output and input data, entropy in SHIFT and NOT operation is the same as ASSIGN.

An example is the implementation of a simple substitution cipher. Although the characters in plaintext are substituted, attackers can easily recover the contents by counting the frequency of symbols used in ciphertext. Therefore, RU generated during ASSIGN, SHIFT, and NOT are all 0, which is in accordance with Equation (3). The length of input or output is defined as W, hence, RU $= |sig_R| - |sig_L| = W - W = 0$.

The Verilog concatenate operator is the open and close brackets, which is used to split or integrate signals. To distinguish those two behaviors, when the close brackets are in transition destination, we call it DECONCATENATE, otherwise we call it CONCATENATE. The DECONCATENATE operation is modeled in Figure 4(c). Assume that the length of the node in transition source is W, and the length of a specific node in transition destination is W'. In the calculation of AR on specific node, the RU produced from this DECONCATENATE operation is $W - W'$.

B. Operations - AND, OR

In the following paragraphs, we discuss the operations whose transition source contains more than one nodes as shown in Figure 4(b). To avoid ambiguity, optimization is performed before the transition source nodes are involved in the operations. Constants and untainted signals in an expression will be calculated and combined first. After that, there are two situations of transition source nodes regarding the security label - a tainted node operates with an untainted node, a tainted node operates with a tainted node. Accordingly, the RU equation should be defined for both cases.

For the case of a tainted node operating with untainted node, attackers observe from the transition destination and know the operation. Therefore, uncertain bits in transition destination nodes depend on the untainted node in the transition source. The uncertainty is in fact the quantity of effective observation of untainted node. Statistically, from assumption 1, expectation of 0 in the untainted node into the AND operation is $\frac{W}{2}$, where W is the length of each input signal. This is the same as 1 in the untainted node to the OR operation. Namely, the secret is hidden because of the 0 value in the untainted node into AND operation, and the 1 value in the untainted node into OR operator. Therefore, the RU equation is $\frac{W}{2}$ for the operations AND/OR.

For the case of tainted node inter-operates with tainted node, since both transition source nodes are secrets, then the RU equation can be derived from Equation (3) directly, i.e., RU $= 2 * W - W = W$. Furthermore, as in Equation (6), the accumulated RU in transition destination is obtained by adding an RU from operation to the accumulated RU from transition source. Then there must be a selection between two ARs from the transition source side. In this situation, the two taint propagations merge on this transition. Because the propagation containing less AR means more suspicious in leaking information, we use the less AR in calculating AR on the transition destination.

C. Operations - XOR

A complex operator can be presented as an equivalent Boolean algebra of basic operators like AND, OR, INV. Accordingly, in hardware, a complicated function cell is composed by combining basic gates. As a result, the RU equation of a complex operator is defined by the more basic operators. For instance, XOR can be made up as in Equation (8).

$$a \quad \text{XOR} \quad b = (a \quad \text{AND} \quad b') \quad \text{OR} \quad (a' \quad \text{AND} \quad b) \quad (8)$$

Note that we only use minimized equivalent Boolean algebra in deducing RU equation. Then, the RU equation for either $(a \quad \text{AND} \quad b')$ or $(a' \quad \text{AND} \quad b)$ can be obtained from the RU equations of basic operations mentioned earlier. Only one part of algebra in the right side of Equation (8) should be considered because there is a close dependency between those two parts. Namely, from the attacker's side, knowing either part, the other one can be inferred completely. Consequently, the RU equation for XOR is $\frac{W}{2}$ for the tainted vs untainted case, and W for the tainted vs tainted case. Also, the results from derivation of equivalent Boolean expression is consistent with the result from Equation (3).

For more complex operators such as ADD, TIMES, etc., the RU equations are obtained by following a similar derivation of RU equations for XOR. Because of the page limits, we cannot show the derivation process of RU equations for all the operators. In Table I, the RU equations for all the frequently used operators in QIF-Verilog are provided.

VI. Experimentation

A. Experiment Setup

In practical applications, the use of the proposed QIF-Verilog is straightforward. As shown in Listing 3, a developer only needs to tag the new type $Taint$ on any signals which may contain sensitive information. In our experiment, the secret key in the top level module of the system is labeled. Then the confidentiality will be checked for those labeled secrets.

To demonstrate the effectiveness of the proposed method supported by the QIF model, we test 11 AES benchmarks

Benchmarks	Threshold	AR-C	Time (s)	AR-T	Time (s)	Detected	Trojan Leakage Payload
AES-T100	318.72	1423	183.4	127.5	232.0	Y	leaks secret key through a covert channel
AES-T200	318.72	1423	184.1	127.5	213.8	Y	leaks secret key through a covert channel
AES-T400	318.72	1423	184.3	128	211.8	Y	transmit the key through an RF signal
AES-T700	318.72	1423	183.4	127.5	210.9	Y	leaks secret key through a covert channel
AES-T800	318.72	1423	184.5	127.5	213.7	Y	leaks secret key through a covert channel
AES-T900	318.72	1423	181.3	127.5	211.2	Y	leaks secret key through a covert channel
AES-T1000	318.72	1423	181.4	127.5	231.8	Y	leaks secret key through a covert channel
AES-T1100	318.72	1423	183.2	127.5	209.7	Y	leaks secret key through a covert channel
AES-T1200	318.72	1423	181.3	127.5	211.0	Y	leaks secret key through a covert channel
AES-T1600	318.72	1423	183.4	128	210.4	Y	transmit the key through an RF signal
AES-T1700	318.72	1423	185.5	128	209.5	Y	transmit the key through an RF signal

Table II: Tests on AES Trojan insertion benchmarks.

Benchmarks	Tainted Signal	Threshold	AR-C	Time (s)	AR-T	Time (s)	Detected	Leakage Path
DES	secret key	139.44	259	1043.5	55.5	1066.8	Y	a covert channel
3DES	one of the secret keys	139.44	259	1048.3	55.5	1071.8	Y	a covert channel
MD5	input data chunk	1247.88	2961	18.5	511.5	22.1	Y	a covert channel
SHA-1 160	input data chunk	79.68	143	32.6	48	36.0	Y	an RF signal
SHA-2 256	input data chunk	79.68	254	82.0	48	86.2	Y	an RF signal
SHA-2 384	input data chunk	79.68	446	329.4	48	341.8	Y	an RF signal
SHA-2 512	input data chunk	79.68	674	335.6	48	343.4	Y	an RF signal

Table III: Tests on leakage path insertion benchmarks.

Benchmarks	F-T (s)	IFT-T (s)	F-T/Total
AES-T1600 (C)	9.2	174.2	5.3%
AES-T1600 (T)	10.6	199.7	5.0%
DES (C)	1039.7	3.8	99.6%
DES (T)	1062.2	4.6	99.6%
3DES (C)	1041.6	6.8	99.4%
3DES (T)	1064.3	7.5	99.3%
MD5 (C)	8.6	9.9	46.5%
MD5 (T)	9.5	12.7	42.8%
SHA-1 160 (C)	30.9	1.7	94.8%
SHA-1 160 (T)	33.7	2.2	93.9%
SHA-2 256 (C)	80.1	1.9	97.7%
SHA-2 256 (T)	83.8	2.5	97.1%
SHA-2 384 (C)	326.0	3.4	99.0%
SHA-2 384 (T)	337.2	4.6	98.7%
SHA-2 512 (C)	332.2	3.5	99.0%
SHA-2 512 (T)	338.8	4.6	98.7%

Table IV: Time consumption in formalization and IFT analysis.

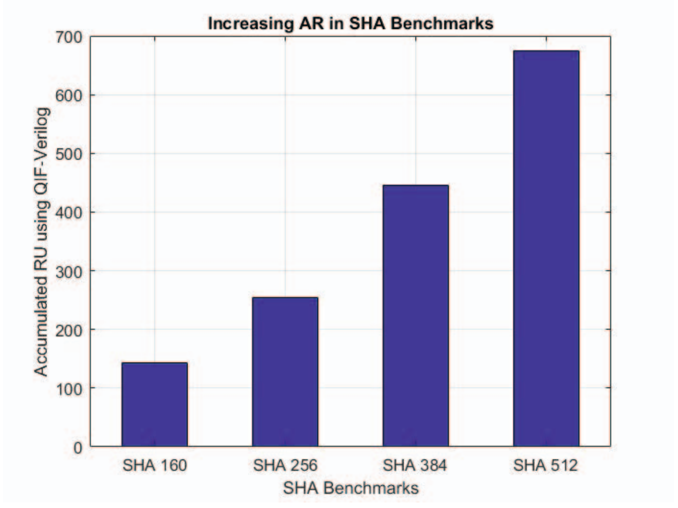

Figure 5: Increasing AR with more strong SHA benchmarks.

from TrustHub [22]–[24] which are fit to the attack model in this paper. Those AES benchmarks are written in Verilog, and contain malicious logic which leaks the secret key from a cryptographic chip running the AES algorithm through different ways. Thus, there are various hardware Trojan types in the benchmarks, serving as good testbed to evaluate the proposed framework in detecting different types of information leakage channels. Note that our approach prevents design mistakes in arbitrary hardware more than Trojans in cryptographic hardware, so the malicious logic in all the AES benchmarks are treated as design faults.

Furthermore, to increase the diversity of the benchmarks, we downloaded seven different Verilog designs from OpenCore [25]. We put those benchmarks in Table III. In these designs, the DES Core takes a standard 56 bits key and 64 bits of data as input and then generates a 64 bits results [26]. The 3DES Core take three standard bits of the key as the DES Core and then generates the 64 bits results. The MD5 benchmark implements a 64-stage pipeline hash function producing a 128-bit hash value [27]. The rest of the four benchmarks belongs to a collection of Secure Hash Algorithm (SHA) cores, which includes implementations of one SHA-1 and three SHA-2 algorithms [28]. Again, we insert the leakage paths to simulate the design faults into the designs. For each benchmark, the leakage path is either a covert channel or an RF signal like the Trojan embedded benchmarks from TrustHub.

Listing 3 aes_128.v using QIF-Verilog

```
module aes_128(clk, state, key, out);
    input         clk;
    input  [127:0] state, key;
    Taint key;
    output [127:0] out;
    reg    [127:0] s0, k0;
    ...
```

To set up the experiment, since the only secret in all the AES benchmarks is 128 bits key, we consider a threshold value of $128 \times 2.49 = 318.72$ for the AES benchmarks depending on Equation (7). Similarly, we label the key from DES as the sensitive secret as well as one of the three keys in 3DES. The threshold value for both the benchmarks is $56 \times 2.49 = 139.44$ For the MD5 and SHAs, the data chunk inputs, 512 bits for the MD5 and 32 bits for the SHA collection, are labeled by $Taint$ in this experiment. Thus, the threshold value is $512 \times 2.49 = 1274.88$ for the MD5 and is $32 \times 2.49 = 79.68$ for the SHA collection, accordingly. In each output port of top model, we check whether the security label $Taint$ exists. If so, the confidentiality of the system is broken and alarm is triggered. Also, a suspicious label propagation path will be provided to the developer. For comparison, we do not set threshold for checking clear benchmarks without leakage paths, such that an Accumulated RU for labeled signal key can be obtained from the output port.

B. Results and Analysis

Table II and Table III show results of our experiments. For each benchmark, we account the Accumulated RU in the output of both the genuine version (with AR-C header) and the Trojan embedded (with AR-T header) version. Corresponding time cost are also recorded. We take the benchmark SHA-1 160 in Table III as an example. The input data chunk of SHA-1 160 core is labeled by $Taint$. An RF signal is inserted as a leakage path shown in the last column. The AR calculated from a genuine version is 143, which is larger than the threshold. The AR from the leakage path insertion version is 48, which is less than the threshold. It means that the leakage path is detected successfully without a false-positive. The entire working procedure takes 32.6 seconds to evaluate the genuine version and 36 seconds for the leakage path insertion version.

Then for all the benchmarks in both tables, the design faults are all detected using the proposed method, and there are no false-positive happen. Through the results, we observe that the accumulated RU from the genuine version benchmark is extremely larger than the leakage paths inserted version, therefore there is a large space to adjust the threshold. It means that the confidentiality between the genuine version and leakage path existing version are clearly distinguishable through using QIF-Verilog. From the detected information leakage types, we summarize that our approach is capable of detecting those Trojans or vulnerabilities whose payload leaks information via logic outputs. From the diversity of benchmarks, the proposed QIF-Verilog is effective to apply to various RTL designs.

From the experimental results in Table IV, the time consumptions of all the benchmarks are demonstrated. Because all of the AES benchmarks from TrustHub are developed from the same genuine AES benchmark, the time costs in formalization stage are similar among the AES benchmarks. Hence in the table, we use one of the AES benchmarks, AES-T1600, as the representation. In the first column, the label (C) stands for the clear/genuine version of the benchmark, while (T) means the leakage-paths inserted version of the benchmark. We divide the total time cost into two stages – a formalization stage which formalizes the QIF-Verilog program to the DFG and an IFT stage which performs the IFT analysis on the DFG. We denote the time cost in the formalization stage as F-T, and in the IFT analysis stage as IFT-T, accordingly. In the last column, we show the ratio of F-T in the total time consumption, which indicates that the formalization stage consumes more time than the IFT stage for most of the benchmarks. Table IV demonstrates that the vulnerability evaluation can be finished in minutes. Even in the worst-case, the time cost is less than 18 minutes. Therefore, the QIF-Verilog is efficient for protecting the confidentiality of large-scale hardware designs.

In the meantime, propagation paths of security labels can be recorded by our solution. If the vulnerability is identified, then the most suspicious path with RU in each transition can be recorded by the designer for further evaluations. For instance, we show the suspicious propagation path in AES-T1600 Trojan embedded benchmark as the following:

AM_Transmission.key→AM_Transmission.SHIFTReg→
AM_Transmission.beep2→AM_Transmission.beeps→
AM_Transmission.MUX_Sel→AM_Transmission.Antena

C. Soundness

The proposed QIF-Verilog provides an evaluation of information leakage vulnerability for a hardware design. From Table II and Table III, the QIF-Verilog has been shown to be effective in detecting vulnerable leakage paths. However, the benchmarks from trusthub are developed for the purpose of evaluating hardware Trojans, which are not same as design vulnerabilities - we treat hardware Trojans as a subset of hardware vulnerability. To prove the soundness of the proposed OIF-Verilog, we utilize the results of benchmarks from the collection of SHA implemented by the same author in the same project [28]. It means that the code style and features are consistent among the four different implementations. The collection includes the cores of one SHA-1 (160) and three SHA-2 (256/384/512).

Both SHA-1 and SHA-2 are versions of the Secure Hashing Algorithm, while SHA-2 is an enhanced successor of SHA-1. The numbers 160/256/384/512 refer to hashes of different bit-length. The shorter the bit-length of hash, like of SHA-1 (160), the more vulnerable it is against attacks, such as the collision and the length extension attacks [29], [30]. In contrast, the SHA benchmark with a longer bit hash is more secure because there are more possible combinations [31]. Therefore, we check the accumulated RU metric of the SHA benchmarks and the results are shown in Figure 5. The Y-axis in the histogram stands for the AR value. We find the metric AR increases with the bit-length of the hash. Thus, we can state the following soundness result: the AR values show improved security with the increase of bit-length of the hash, thus our proposed method is evaluating security of design correctly.

VII. Conclusion

In this paper, QIF-Verilog, a secure language extended from Verilog, is proposed to protect the confidentiality of a large-scale secure hardware design. Given that the preliminary language based secure solutions either lead to a high overhead or are inflexible for practical use, our solution solves these issues by introducing a new type $Taint$ and applying QIF model to Verilog semantics. Specifically, an accumulated RU is generated through calculating entropy to quantify the leakage of labelled secrets in the hardware design. In such a way, the information leakage vulnerability can be detected by identifying the security label in output without hardware overhead. The results show that the approach is effective against those vulnerabilities that leak sensitive information via logical outputs. Also, our proposed approach is efficient and practical for protecting large-scale hardware designs.

In the future, more properties and features will be considered in the QIF-Verilog. The integrity property will be considered to identify vulnerabilities caused by malicious modifications. Also, We will extend the QIF-Verilog for general purpose hardware such as microprocessors. Besides the micro-controller architecture, the storage like program memory will also be formalized. A more systematic security evaluation will be given by the QIF-Verilog.

Acknowledgements

This work is partially supported by Army Research Office (W911NF-17-1-0477), Semiconductor Research Corporation (2014-TS-2554), and Cisco.

References

[1] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *IEEE Symposium on Security and Privacy*, 2007, pp. 296–310.

[2] Y. Jin and Y. Makris, "Hardware Trojan detection using path delay fingerprint," in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 51–57.

[3] K. Xiao, A. Nahiyan, and M. Tehranipoor, "Security rule checking in ic design," *Computer*, vol. 49, no. 8, pp. 54–61, 2016.

[4] Y. Jin, X. Guo, R. G. Dutta, M.-M. Bidmeshki, and Y. Makris, "Data secrecy protection through information flow tracking in proof-carrying hardware ip—part i: Framework fundamentals," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 10, pp. 2416–2429, 2017.

[5] M.-M. Bidmeshki, X. Guo, R. G. Dutta, Y. Jin, and Y. Makris, "Data secrecy protection through information flow tracking in proof-carrying hardware ip—part ii: Framework automation," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 10, pp. 2430–2443, 2017.

[6] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 1997, pp. 129–142.

[7] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 99–110, 2012.

[8] 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.

[9] 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," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 109–120.

[10] 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 *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 97–112.

[11] G. Smith, "On the foundations of quantitative information flow," in *International Conference on Foundations of Software Science and Computational Structures*. Springer, 2009, pp. 288–302.

[12] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 286–296.

[13] P. Mardziel, M. S. Alvim, M. Hicks, and M. R. Clarkson, "Quantifying information flow for dynamic secrets," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 540–555.

[14] 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.

[15] SecVerilog, "Secverilog documentation," http://www.cs.cornell.edu/projects/secverilog/.

[16] S. A. M'rio, K. Chatzikokolakis, C. Palamidessi, and G. Smith, "Measuring information leakage using generalized gain functions," in *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE, 2012, pp. 265–279.

[17] J. L. Massey, "Guessing and entropy," in *Information Theory, 1994. Proceedings., 1994 IEEE International Symposium on*. IEEE, 1994, p. 204.

[18] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*. IEEE, 2005, pp. 255–269.

[19] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for efficient control of timing channels," 2014.

[20] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[21] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2015, pp. 451–460.

[22] *https://www.trust-hub.org/*.

[23] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE, 2013, pp. 471–474.

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

[25] *https://opencores.org/*.

[26] U. Rudolf, "https://opencores.org/project/des/," 2009.

[27] L. John, "https://opencores.org/project/md5_pipelined/," 2018.

[28] Marsgod, "https://opencores.org/project/sha_core/," 2018.

[29] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. P. Bianco, and C. Baisse, "Announcing the first sha1 collision," *Google Security Blog*, 2017.

[30] T. Duong and J. Rizzo, "Flickr's api signature forgery vulnerability," *Tech. Rep.*, 2009.

[31] National Institute of Standards and Technology, "Descriptions of sha-256, sha-384, and sha-512."