Hardware Information Flow Tracking

WEI HU, Northwestern Polytechnical University, China
ARMAITI ARDESHIRICHAM, University of California, San Diego, USA
RYAN KASTNER, University of California, San Diego, USA

Information flow tracking (IFT) is a fundamental computer security technique used to understand how information moves through a computing system. Hardware IFT techniques specifically target security vulnerabilities related to the design, verification, testing, manufacturing, and deployment of hardware circuits. Hardware IFT can detect unintentional design flaws, malicious circuit modifications, timing side channels, access control violations, and other insecure hardware behaviors. This article surveys the area of hardware IFT. We start with a discussion on the basics of IFT, whose foundations were introduced by Denning in the 1970s. Building upon this, we develop a taxonomy for hardware IFT. We use this to classify and differentiate hardware IFT tools and techniques. Finally, we discuss the challenges yet to be resolved. The survey shows that hardware IFT provides a powerful technique for identifying hardware security vulnerabilities as well as verifying and enforcing hardware security properties.

CCS Concepts: • **Security and privacy → Logic and verification**; **Information flow control**; **Formal security models**; • **General and reference → Surveys and overviews**.

Additional Key Words and Phrases: Hardware security, information flow security, information flow tracking, security verification, formal method, survey.

ACM Reference Format:
Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware Information Flow Tracking. *ACM Comput. Surv.* 0, 0, Article 00 (January 2021), 37 pages. https://doi.org/10.0000/0000000.0000000

1 INTRODUCTION

A core tenet of computer security is to maintain the confidentiality and integrity of the information being computed upon. Confidentiality ensures that information is only disclosed to authorized entities. Integrity maintains the accuracy and consistency of information – verifying that it is correct, complete, and not modified in any unauthorized manner. Understanding how information flows throughout a computing system is crucial to determine if that system operates securely with respect to confidentiality and integrity.

Information flow tracking (IFT) is a security technique that models how information propagates as a system computes. It labels data objects with a tag to denote security classes, which are assigned different meanings depending on the type of security property under analysis. IFT updates the tags as the data is computed upon, and verifies information flow properties by observing the state of the tags.

Authors' addresses: Wei Hu, Northwestern Polytechnical University, 127 Youyi Xi Lu, Xi'an, Shaanxi, 710072, China, weihu@nwpu.edu.cn; Armaiti Ardeshiricham, University of California, San Diego, 9500 Gilman Dr., La Jolla, CA, 92093, USA, aardeshi@ucsd.edu; Ryan Kastner, University of California, San Diego, 9500 Gilman Dr., La Jolla, CA, 92093, USA, kastner@ucsd.edu.

IFT can reason about the confidentiality and integrity properties. It verifies confidentiality by determining if secret information can ever be disclosed to an unclassified location. An example confidentiality property states that information related to a secret key should not leak outside a restricted memory space. IFT can also verify data integrity properties by not allowing untrusted entities to operate on trusted information. An example integrity property states that a user level program can never modify the system configuration data.

IFT has been used to enforce security properties across the computing system stack including operating systems [36, 78, 161], programming languages [128], distributed systems [105, 162], and cloud computing [6]. It can detect and prevent a wide range of software security issues including buffer overflow, memory corruptions, SQL injection, formatted string attack, cross-site scripting attacks, and malware [1, 37, 102, 109, 139]. IFT has been applied to verify the functional correctness and security of complex software systems like the *seL4* secure operating system (OS) kernel [74, 104]. More recently, hardware IFT techniques have been used to detect and assess security threats related to hardware design vulnerabilities [65, 137], insecure debug modes [16, 60, 124], timing channels [2, 4, 30, 64, 163], and hardware Trojans [45, 53, 108, 154]. A survey on security verification of hardware/software systems cited information flow analysis as the most commonly used technique for modeling and reasoning about security [27].

This article introduces a hardware IFT taxonomy, which is used to survey, classify, and compare hardware IFT techniques. We start by discussing the initial information flow model put forth by Denning [31, 32]. We identify the distinguishing characteristics of Denning's information flow model including the *security property* enforced, *level of abstraction*, *operator precision*, and *verification technique*. We use this to create a taxonomy to help compare and contrast the hardware IFT techniques. Figure 1 shows a taxonomy which consists of the following elements:

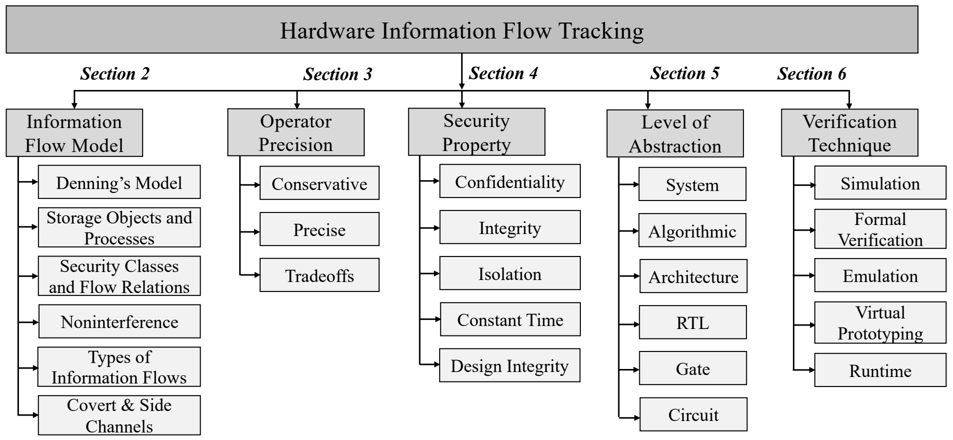


Fig. 1. A taxonomy for hardware IFT. We start by describing the characteristics of an information flow model and define basic terminology. After that, we expand on different ways to classify the IFT techniques. The precision of the class combining operator and the type of security property under consideration play an important role in defining the characteristics of the IFT technique. Another distinguishing characteristic is the abstraction level of the technique's information flow model. Finally, IFT techniques enable different forms of verification, which provides another method of differentiation.

- **Operator Precision:** How does the class combining operator update security classes?
  - **Conservative:** IFT operator uses a least upper bound.
  - **Precise:** IFT operator considers the effect of data values.
  - **Tradeoffs:** IFT operator performs tradeoffs between precision and computational complexity.

- **Security Property:** What types of security properties are supported?
  - **Confidentiality:** IFT technique prevents leakage of sensitive information.
  - **Integrity:** IFT technique prohibits overwriting of trusted data by an untrusted entities.
  - **Isolation:** IFT technique prevents communication between two entities of different trust.
  - **Constant Time:** IFT technique captures information flows through runtime variations.
  - **Design Integrity:** IFT technique detects malicious information flows caused by undocumented design modifications.
- **Level of Abstraction:** What is the level of abstraction of the information flow model?
  - **System:** IFT technique considers system level flows.
  - **Algorithmic:** IFT technique is deployed during high-level synthesis.
  - **Architecture:** IFT technique is deployed at the instruction set architecture (ISA) level.
  - **RTL:** IFT technique targets register transfer level (RTL) design.
  - **Gate:** IFT technique considers a gate-level netlist.
  - **Circuit:** IFT technique targets analog and mixed-signal hardware designs.
- **Verification Technique:** Which verification techniques are supported?
  - **Simulation:** IFT technique uses simulation tools to determine information flows.
  - **Formal verification:** IFT technique employs formal methods to verify security properties.
  - **Emulation:** IFT technique allows for hardware emulation of information flow behaviors.
  - **Virtual prototyping:** IFT technique creates a software version of hardware to measure information flows.
  - **Runtime:** IFT technique dynamically tracks information flows during runtime.

We use this taxonomy to classify and describe different hardware IFT techniques. This helps us discuss the relative advantages, weaknesses, and drawbacks of these techniques. We attempt to draw a clear picture of the past and future of hardware IFT research with a focus on its capability for developing new hardware security tools.

The remainder of this article is organized as follows. In Section 2, we discuss the core concepts of information flow, covering the fundamental elements in Denning's information flow model [32], the classical noninterference model [43], the different types of flow relations as well as basics of covert and side channels. Section 3 formalizes the concept of class combining operator precision and discusses IFT precision and computational complexity tradeoffs. In Section 4, we describe the different types of security properties that can be modeled and enforced under the information flow framework. Sections 5 characterizes hardware IFT techniques by their level of abstraction while Section 6 elaborates on the various mechanisms for performing information flow security verification. We discuss potential research vectors and challenges in Section 7, and conclude the article in Section 8.

2 INFORMATION FLOW

Information flow tracking (or information flow control) verifies or enforces that no unauthorized flow of information is possible. IFT techniques work by labeling storage objects with a security class, tracking those labels as their data is computed upon, and enforcing information flow policies to understand, verify, and/or control the movement of information in the system. Simple examples of policies include: 1) `confidential` information should never leak to an `unclassified` location and 2) `untrusted` information should never overwrite a `trusted` memory component.

Denning developed the earliest models for *information flow* [31, 32], which enables one to define the allowable flows of information between storage objects $\mathcal{N}$ using security classes $\mathcal{SC}$ and flow relations $\rightarrow$. Processes $\mathcal{P}$ operate on those storage objects. And a class combining operator $\oplus$ provides a relationship between security classes. We use Denning's information flow model to discuss and compare different hardware IFT techniques.

An information flow model $\mathcal{FM}$ is defined by $\mathcal{FM} =< \mathcal{N}, \mathcal{P}, \mathcal{SC}, \oplus, \rightarrow>$ where:

- $\mathcal{N} = \{a, b, \cdots\}$ is a set of *storage objects*. Elements of $\mathcal{N}$ can vary depending on the level of abstraction under consideration, e.g., files, segments, registers, and flip-flops.
- $\mathcal{P} = \{p, q, \cdots\}$ is a set of *processes* where information flows through, e.g., functions, arithmetic operations and Boolean gates.
- $\mathcal{SC} = \{A, B, \cdots\}$ is a set of *security classes* corresponding the security classification of data objects, e.g., `secret` and `unclassified`. The security class associated with the information stored in an object a can either be bound statically or updated dynamically with the flow of information. These are also commonly called *labels* or *tags*.
- The *class combining operator* $\oplus$ takes a pair of security classes and calculates the resulting security class, i.e., $\mathcal{SC} \times \mathcal{SC} \mapsto \mathcal{SC}$.
- A *flow relation* $\rightarrow$ is defined on a pair of security classes. It is a "can flow" relation on $\mathcal{SC}$, i.e., $\rightarrow \subseteq \mathcal{SC} \times \mathcal{SC}$. Information flows from class A to class B whenever information associated with A could affect the value of data object associated with B. In other words, there exists a situation where changing the value of object associated with A leads to a change in the value of object associated with B. We write $A \rightarrow B$ if and only if information in class A is allowed to flow to class B.

We adopt this classic notation as much as possible throughout this article. Our goal is to classify the work in hardware IFT using this notation so that it can provide a common language to compare and contrast the different techniques and methodologies. In the following subsections, we elaborate on the elements of this information flow model. Section 2.1 describes *storage objects* and *processes* from a hardware viewpoint. Section 2.2 discusses relevant information related to the *flow relations* and *security classes*. Section 2.3 introduces the idea of *noninterference*. Section 2.4 categorizes and illustrates the different types of *information flow relations*. Section 2.5 covers the basics of *covert* and *side channels*.

2.1 Storage Objects and Processes

An *object* is a container of information and a *process* describes computations performed upon those objects. Storage objects and processes vary depending on the security policies and level of abstraction. The initial work by Denning [31, 32] largely considered the security of software processes or programs. Much of it focused on security at the OS level, and thus objects were things like files, directories, and users. Hardware storage objects are constructs defined in hardware description languages (HDLs) and their corresponding logical representations and physical implementations.

In hardware design, storage objects are defined using HDL types that can take or store values, e.g., input and output ports, internal wires and signals, registers, flip-flops, and memory blocks. It is often convenient to consider objects as any stateful elements in the hardware, i.e., to ignore wires and signals and focus on registers, flip-flops, and memory blocks. These objects are operated upon by processes $\mathcal{P}$ that can be defined as register transfer operations, finite state machine (FSM) actions, combinational functions, and other common hardware models of computation.

Unfortunately, "process" is an overloaded term since `process` is a defined keyword in VHDL that denotes a set of sequential statements. The security notion of a process ($\mathcal{P}$) covers a much broader scope; it refers to any arithmetic, logical or analog hardware component that operates on input data objects and produces a resulting object. Examples of hardware process $\mathcal{P}$ include arithmetic, logical, and circuit-level operations.

2.2 Security Classes and Flow Relations

Information flow policies define allowable relationships between data objects. To do this, IFT associates each object with a security class, i.e., gives objects a *security label* or *security tag*. Objects define the functional state of the system; their security labels determine the system security state.

The flow relation operator $\longrightarrow$ defines an allowable flow of information on two security classes. $\mathcal{SC} = \{A, B\}$, $A \longrightarrow B$ states that information from security class A is allowed to flow to security class B. We can also denote the lack of an information flow with the operator $\nrightarrow$, e.g., $B \nrightarrow A$ states that information in security class B should never flow to security class A.

Security practitioners often use a *lattice* to describe flow relations. For example, Denning defines information flow policies as a finite lattice [32]. A lattice is in the form of $\mathcal{L} = \{\mathcal{E}, \sqsubseteq\}$, where $\mathcal{E}$ is the set of elements and $\sqsubseteq$ is a partial order relation defined on $\mathcal{E}$. The tuple $\{\mathcal{E}, \sqsubseteq\}$ constitutes a *lattice* if there exists a *least upper bound* element and a *greatest lower bound* element for any $A, B \in \mathcal{E}$ in the element set. The class combining operator $\oplus$ is also a least upper bound operator in that for all $A, B, C \in \mathcal{E}$:

$$\begin{aligned} &A \longrightarrow A \oplus B \text{ and } B \longrightarrow A \oplus B \\ &A \longrightarrow C \text{ and } B \longrightarrow C \implies A \oplus B \longrightarrow C \end{aligned} \tag{1}$$

The greatest lower bound operator $\odot$ is defined as $A \odot B = \oplus LSet(A, B)$, where $LSet(A, B) = \{C \mid C \longrightarrow A \text{ and } C \longrightarrow B\}$ is the set of security classes from which information can flow to both A and B.

We usually consider lattices with a finite number of elements. Let $\mathcal{E} = \{A_1, A_2, \cdots, A_n\}$. The least upper bound (also called *maximum element*, denoted as `high`) and the greatest lower bound (also called *minimum element*, denoted as `low`) on the lattice are defined as follows:

$$\begin{aligned} \texttt{high} &= A_1 \oplus A_2 \oplus \cdots \oplus A_n \\ \texttt{low} &= A_1 \odot A_2 \odot \cdots \odot A_n \end{aligned} \tag{2}$$

An information flow policy/property can be modeled using a *security lattice* $\mathcal{L} = \{\mathcal{SC}, \sqsubseteq\}$, where $\mathcal{SC}$ is a set of security classes and $\sqsubseteq$ is the partial order defined on $\mathcal{SC}$. Given any two security classes $A, B \in \mathcal{SC}$, we say A is `lower` than B (or B is `higher` than A) when $A \sqsubseteq B$.

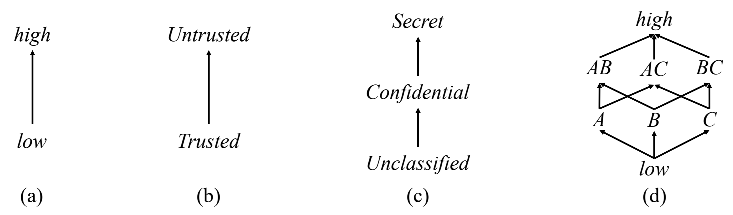


Fig. 2. Example security lattice structures. The lattice defines allowable information flows. Any operations that lead to an unallowable flow (e.g., allowing `Untrusted` information to flow to a `Trusted` data object or `Confidential` information to flow to an `Unclassified` data object) result in a security violation. Information flow is allowed to flow upward along the arrow in the security lattice; downward flows of information are not permitted.

Figure 2 provides four examples of lattices commonly used in modeling security. Figure 2 (a) shows the `high` and `low` security lattice that we mentioned earlier. Here, the security class set has two elements $\mathcal{SC}$ = {`high`, `low`}. The partial order (reflected by the arrow) defines allowable flows of information among different security classes, e.g., `low` $\sqsubseteq$ `high` indicating that `low` information is allowed to flow to a `high` data object.

Figure 2 (a) to (c) are linear security lattices, where any two security classes are ordered or comparable. There are also more complex lattice structures with partial ordering. Figure 2 (d) gives an example of such non-linear security lattice, which contains non-comparable security classes, e.g., security classes `A` and `B` as well as `A` and `BC`. Information flow between security classes without partial order defined by the lattice should be prohibited.

In practice, most policies use a two-element lattice like in Fig. 2 (a). This allows the modeling of security properties related to confidentiality and integrity as `low` $\sqsubseteq$ `high`. In cases where multi-level security (MLS) needs to be enforced, oftentimes we employ conservative label operators (e.g., the least upper bound or greatest lower bound operators) to determine the security class that the output of a process should take. More precise security class updating rules expand the two-level technique to account for the influence of the data values on security class operation [54, 55].

2.3 Noninterference

Noninterference is a strict MLS model proposed by Goguen and Meseguer [43]. It creates a model of information flow by modeling all system inputs, outputs, and state as either `high` or `low`. It states that any changes in `high` inputs shall not result in changes in `low` outputs. That is, `low` objects can learn nothing about `high` data. Another way of stating this is that a computer system produces the same `low` outputs for any set of `low` inputs regardless of the `high` inputs. That is, it is impossible to learn any information about the `high` values by changing the `low` inputs. Equivalently, the computer system responds exactly the same to any input sequence of `low` values regardless of the `high` values.

Noninterference provides a notion of strong, end-to-end security properties [130]. That is, it tracks the release and subsequent transmissions of the data to ensure that they adhere to the desired security properties. It extends the notion of access control [33, 80, 156], which strictly limits accesses to objects in the system. These lists enable restrictions on who can access an object; yet once it is released, its propagation is not restricted, and this information can be subsequently transmitted in an insecure manner.

Definitions of noninterference often rely on semantic models of program execution. Early work in this space includes Cohen's work using the notion of strong dependency [23, 24] and McLean's definition of noninterference using trace semantics [101]. Volpano *et al.* formalize Denning's lattice approach as a type system viewable as a form of noninterference [150]. Many follow-on works extend the expressiveness of the model and the typing system to handle concurrency, nondeterminism, synchronization and exceptions; for more information, see Sabelfield and Myers' excellent survey on language-based information flow security [128].

Much of this early work on noninterference was related to accessing user data and isolating programs running on shared computers. Thus, it largely dealt with files, resources, and users as objects and took a very "software" or operating system centric view. Regardless, the general notion of noninterference translates well to hardware. Perhaps the most straightforward example is the SecVerilog project [163], which employs the notion of noninterference using semantic models from Verilog. SecVerilog adds security labels to the Verilog language and extends previous security typing languages to hardware design. The challenge is accounting for all the eccentricities of the hardware models of computation, and in particular of vast use of concurrency. Regardless, the general idea that `low` objects should not be able to infer any information about the `high` objects still holds and can be translated to hardware.

A major challenge is that the number of objects in hardware is typically much larger than in software. Hardware objects may include the entire memory space, registers, and other low-level state – any stateful storage element could be considered. Of particular note is that what is considered an object varies depending on the level of abstraction. For example, objects at the operating system level are files and users. ISA-level objects are visible programmer state, e.g., registers, memory, and some program state (PC, status and configuration registers). At the RTL level, objects include intermediate registered values and state related to control flow. Objects at the gate level would likely include every flip-flop which would include all registers (programmer visible or not) and the state of different controllers (i.e., finite state machine bits). Enforcing noninterference across all of these objects can be challenging. Yet, regardless of the number of objects, the notion of noninterference remains true – any information related to `high` objects should not flow to `low` objects.

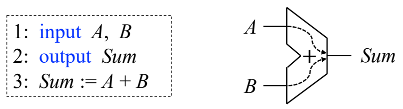
2.4 Types of Information Flow Relations

While flow relations provide a way to specify security policies related to whether or not information is allowed to be transferred between objects, they do not strictly define the method to determine if there is a flow of information. There are many different ways in which information can be transmitted. Understanding the different types of information flow relations and how they manifest themselves in hardware is an important first step for precisely measuring all flows of information and further tightly enforcing the security policies.

Information can flow through functional (e.g., Boolean gates, arithmetic operators, and transactions) and physical (e.g., power, electromagnetic, and thermal) channels. A major difference between functional and physical flows is how information is encoded and transmitted. Functional flows relate to information transmitted through the specified design functionality. The simplest example is directly writing data from one object to another – the functionality here states that information flows through these objects. Physical information flows relate to the physical process of performing the computation. This could be the time that an operation takes, e.g., the time to receive data from a cache or to compute some operation whose execution time depends on the data involved in the computation (e.g., modular exponentiation [77]). Additionally, any circuit switching activity results in power consumption, electromagnetic radiation, and thermal emission. Each of these can be used to infer what is being computed, and thus creates a flow of information.

In this article, we primarily focus on functional information flows, which traditionally have been further categorized as *explicit* and *implicit*. In the following, we illustrate the difference between explicit and implicit flows. We also discuss a special type of implicit flow called *timing flow*, which can lead to leakage of sensitive information.

2.4.1 ***Explicit Flow***. *Explicit flow* is associated with the direct movement of data. Thus, it is also called *data flow*. As shown in Fig. 3, information flows explicitly from the source operand(s) to the destination operand (from A and B to Sum in this example) when the expression $Sum := A + B$ is evaluated.

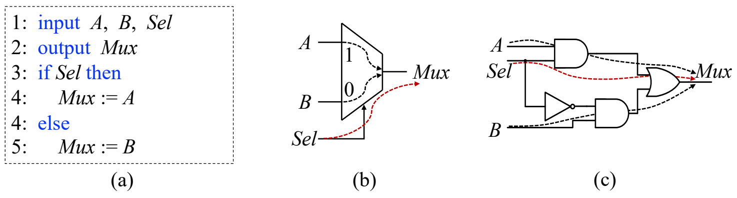


```
input A, B
output Sum
Sum := A + B
```

Fig. 3. A simple example of explicit flow. The data from A and B explicitly flow to Sum when the add operation is executed.

2.4.2 ***Implicit Flow***. *Implicit flow* is a more subtle type of information flow that is caused by context-dependent execution such as conditional operations. Here, information can flow between two objects even when there is no explicit data assignment. Consider the example shown in Fig. 4 (a), which implements a simple if/else control structure.

It is clear that both A and B flow explicitly to Mux through the statements. The more challenging question is whether there is an information flow between Sel and Mux. There is obviously no explicit flow. However, it is possible to infer the value of Sel depending on what is written into Mux, i.e., whether Mux is assigned the value from A or B. Thus,



```
input A, B, Sel
output Mux
if Sel then
    Mux := A
else
    Mux := B
```

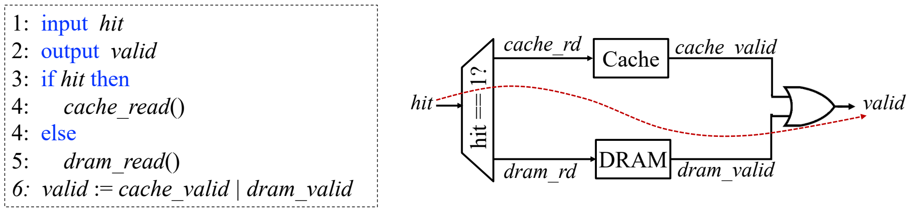
Fig. 4. A simple example of implicit flow. (a) The if/else hardware control structure causes a flow of information from Sel to Mux even though there is no direct assignment of the Sel data to Mux. As the design is synthesized to RTL (b) and gate level (c), the implicit flow becomes explicit as the multiplexer in (b) and the gates in (c) now denote an explicit flow between Sel and Mux.

there is an implicit information flow between *Sel* and *Mux*. Information flows implicitly from the conditional variable *Sel* to *Mux*, since the final value of *Mux* depends on the value of *Sel*.

These implicit flows start to become more like explicit flows at lower levels of abstraction. For example, in Fig. 4 (b) and (c), the if/else statement from Fig. 4 (a) is synthesized into a multiplexer, which is a three input (*A*, *B*, and *Sel*) one output (*Mux*) function block. As synthesis progresses, the multiplexer is further decomposed into gates which again implements the same function. Thus, the implicit flow from Fig. 4 (a) becomes an explicit flow after RTL (Fig. 4 (b)) and logic synthesis (Fig. 4 (c)). As RTL conditional branch statements are synthesized to multiplexers and even smaller primitive Boolean gates, the information flow from *Sel* to *Mux* is indistinguishable from those from *A* or *B* to *Mux*. In other words, all functional information flows are explicit at the gate level. At the same time, it becomes harder to decouple the different types of flows as the level of abstraction is lowered. For example, assume a security policy needed to differentiate between explicit and implicit flows. This would be more challenging to distinguish at the gate level.

2.4.3 *Timing Flow*.

A *timing flow* is a special type of implicit flow where information flows through timing-related design behavior. Figure 5 shows a simple example of cache timing flow. The value of the cache signal hit affects the time at which the $valid$ signal is asserted. It is not a functional flow because $valid$ will eventually be asserted regardless of the value of hit. Yet, by observing the time at which $valid$ is asserted, an attacker can infer the value of hit. Thus, hit flows to $valid$ through a timing channel.



```
input hit
output valid
if hit then
    cache_read()
else
    dram_read()
valid := cache_valid | dram_valid
```

Fig. 5. A cache timing flow. The signal hit flows to the signal $valid$ through a timing channel, i.e., an attacker can determine information about hit by observing when $valid$ is asserted.

An information flow measurement may also be a mix of functional and timing. Consider the RSA implementation via repeated square-and-multiply that was targeted in Kocher's timing attack [77]. There is a functional flow from *key* to ciphertext since the value of key directly affects the encryption result. However, there is also a timing flow from *key* to *cipher* in this implementation since the individual key bits determine the amount of time needed to encrypt a given message. It is possible to distinguish between timing flows and functional flows using hardware IFT [2, 114].

2.5 Covert and Side Channels

A *covert channel* is a transfer of information using an unintended source. The Orange Book classifies two types of covert channels: storage and timing channels [100]. *Storage covert channels* transmit information to a "storage location" (e.g., disk sectors) from parties in different security levels that are not allowed to communicate. *Timing covert channels* transmit information through the usage of system resources, which are observable by another process.

Lampson defines a covert channel as a channel "not intended for information transfer at all, such as the service program's effect on system load" [79]. Sabelfeld and Myers followed this notion and characterized covert channels as ways to transfer information within a computing system using mechanisms not intended to function as information channels. They classify covert channels into several categories, including implicit flows, termination channels, timing channels, probabilistic channels, resource exhaustion channels, and power channels [128]. Covert channels can be a security threat depending on the type of information they allow attackers to observe about the computing system, e.g., execution time, power consumption, resource usage and probability distribution of observable data.

By comparison, a side channel is a leakage of information through non-functional (often physical) characteristics, e.g., execution time, power consumption, EM radiation, and acoustic emissions. Kocher pioneered the use of side channel attacks to recover the cryptographic key via statistical analysis on time and power trace measurements [76, 77]. *Meltdown* [88], *Spectre* [75], and *Foreshadow* [14, 155] leverage architectural level security flaws to load kernel information into cache, and then use timing channel attack to retrieve protected information.

An interesting consequence of modeling information flow at the hardware level, is that it provides cycle-accurate timing information enabling timing channels (or lack thereof) to be modeled and formally verified using IFT [2, 4]. Section 4 describes this in more detail.

Covert and side channels exploit similar phenomenon, e.g., timing, power consumption, and thermal radiation. The major difference between a covert and a side channel is intent. A covert channel involves one party attempting to transfer information to another; a side channel is usually unintentional, but is often exploited as a vulnerability.

3 OPERATOR PRECISION

IFT techniques can use different class combining operators (or label propagation policies) to calculate the security class of the output upon input security labels. The type of operator employed affects both the precision of the IFT in measuring information flows, and the computational complexity of the measurement. This section categorizes IFT techniques by operator precision, and discusses precision and complexity tradeoffs.

3.1 Precision of IFT

The precision of an IFT technique reflects its ability to accurately measure all (and only) existing information flows in hardware designs. Let O be an n-input function denoted as $O = f(I_1, I_2, \cdots, I_k, \cdots, I_n)$. A precise IFT technique indicates a flow from I_k to O *if and only if* the value of input I_k $(1 \leq k \leq n)$ has an effect on that of O, i.e., a change in the value of I_k will result in a change in O.

$$I_k \rightarrow O \Leftrightarrow f(I_1, I_2, \cdots, I_k, \cdots, I_n) \neq f(I_1, I_2, \cdots, I_k^{'}, \cdots, I_n) \tag{3}$$

Precise IFT techniques will accurately model the information flow behaviors of hardware designs, while imprecise techniques may either indicate non-existent information flows (i.e., false positives) or miss actual information flows (i.e., false negatives). The false positives in information flow measurement correspond to information flow behaviors that would not actually happen. Most imprecise IFT techniques tend to introduce false positives, which are safe but may lead to conservative design decisions, i.e., a perfectly secure design may be verified as insecure. However, it should be noted that false negatives also arise, often due to incomplete analysis, e.g., when an information flow security property cannot be verified within reasonable amount of time and resource.

3.2 Imprecise IFT

Early hardware IFT techniques all tend to employ the conservative least upper bound operator in Denning's information flow framework to calculate the output security class and determine the flow of information [13, 139]. These techniques assume that information will always propagate from the inputs to the outputs that are driven by them. In other words, information flows are independent of the functionality of the component and input conditions; the output will take a `high` label as long as any input is `high`. Let $\mathcal{L}(I_1), \mathcal{L}(I_2), \cdots, \mathcal{L}(I_n)$ be the security labels of the inputs $I_1, I_2, \cdots, I_n$ respectively and $\mathcal{L}(O)$ be the security label of output O. This conservative label propagation policy can be formalized as Equation (4), where the $\oplus$ symbol represents the least upper bound operator. Imprecise IFT techniques employing this label propagation policy have a computational complexity of $O(2^n)$.

$$\mathcal{L}(O) = \mathcal{L}(I_1) \oplus \mathcal{L}(I_2) \oplus \cdots \oplus \mathcal{L}(I_n) \tag{4}$$

The label propagation policy shown in Equation (4) can be conservative for certain hardware components according to Equation (3). Consider the two-to-one multiplexer (MUX-2); whether or not an input propagates to the output depends on the value of the select line.

Although imprecise IFT techniques may lead to conservative verification results, they usually allow a quick profile of potential information flow security vulnerabilities. This can be useful for identifying security violations that occur under rare conditions, e.g., a hardware Trojan that leaks information only when triggered [53]. However, the verification performance benefits that come at the cost of loss in precision may later be counteracted by additional efforts needed to exclude false alarms, e.g., a disabled debug port that had access to critical memory locations [52].

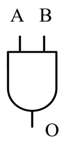
3.3 Precise IFT

Precise IFT techniques take into account both the functionality of a hardware component and the input values when determining flow relations. This label propagation policy can be formalized as a function of both the inputs and their labels as shown in Equation (5). Thus, the complexity of precise IFT techniques is $O(2^{2n})$.

$$\mathcal{L}(O) = f(I_1, I_2, \cdots, I_n, \mathcal{L}(I_1), \mathcal{L}(I_2), \cdots, \mathcal{L}(I_n)) \tag{5}$$

Gate level information flow tracking (GLIFT) [146] is the first well-established precise IFT technique. Figure 6 shows the difference in class combining operations with/without considering data values.

Figure 6 (b) and (c) indicate a subset of partial flow relations and class combining operations without and with consideration of data values, respectively. The entries in red text reveal the differences between these two rule sets. In Fig. 6 (b), the least upper bound operator is used for class combining and thus `high` information will always flow from B to O. However, the values of the input data can in some cases determine the output, and could even prevent the flow of information from another input. Take the second and third rows in Fig. 6 (c) for example. The (`low`, `0`) input in A dominates the output to be (`low`, `0`). In such cases, the `high` information in input B does not have any effect on the output and thus cannot flow to O.



(a)

A	B	O	Flow →
(low, 0)	(high, 0)	(high, 0)	high → L(O)
(low, 0)	(high, 1)	(high, 0)	high → L(O)
(low, 1)	(high, 0)	(high, 0)	high → L(O)
(low, 1)	(high, 1)	(high, 1)	high → L(O)

(b)

A	B	O	Flow →
(low, 0)	(high, 0)	(low, 0)	low → L(O)
(low, 0)	(high, 1)	(low, 0)	low → L(O)
(low, 1)	(high, 0)	(high, 0)	high → L(O)
(low, 1)	(high, 1)	(high, 1)	high → L(O)

(c)

Fig. 6. Examples of flow relation and class combining operation. (a) Two-input AND gate. (b) Flow relation and class combining operation without considering data values. (c) Flow relation and class combining operation with consideration of data values.

Oberg and Hu *et al.* have formalized precise IFT logic for Boolean gates under the notion of GLIFT [57, 112]. In a successive work, they performed a formal analysis on the complexity of the precise hardware IFT logic generation problem and revealed the impreciseness that can arise when tracking information flows in a constructive manner [58].

Ardeshiricham *et al.* proposed RTLIFT [3], a precise hardware IFT technique at the RTL. The major challenges are handling complex conditional branch structures such as if-else, case and loop statements. However, RTLIFT allows more effective handling of timing flows modeled as conditional register updates controlled by tainted values [2].

Although RTLIFT has observed a magnitude of improvement over GLIFT in terms of verification performance and scalability, precise IFT can inevitably be expensive for large hardware designs since it is inherently an NP-complete problem [58]. Thus, it is desirable to derive practical IFT methods that strike a balance between performance and precision in order to achieve verification goals with a more reasonable cost in terms of time and resource usage..

3.4 Precision and Complexity Tradeoffs

Precision and complexity are two contradictory factors for evaluating hardware IFT techniques [52]. We have to tradeoff one for the other like we usually do for area and performance in logic synthesis.

Switching circuit theories provide insightful explanations about the impreciseness of hardware IFT techniques. Oberg and Hu *et al.* have identified the root cause of false positives in IFT analysis as static logic hazards [57, 112]. They used a MUX-2 example to demonstrate that false positives arise when there is either static-0 or static-1 logic hazard.

Hu *et al.* associate the source of imprecision with internal variable correlations caused by reconvergent fanouts [58]. They also formally proved that precise hardware IFT logic generation is an NP-complete problem. To overcome the exponential scale complexity of the precise IFT logic generation problem, constructive as well as heuristic methods were proposed to create imprecise tracking logic in polynomial time at the cost of loss in precision.

In [52], Hu *et al.* investigated the effect of logic synthesis optimizations on precision of IFT. This work reveals that the false positives of hardware IFT hide behind unreachable satisfiability don't care conditions in digital circuits. It provides an efficient approach to deriving different version of hardware IFT logic with variable precision and complexity for Boolean gates. The observation is that precise IFT logic more accurately measures information flows because it considers the value of variables in label propagation. We move towards the less precise by gradually ignoring the inputs to relax precision. Figure 7 illustrate this process using the two-input AND (AND-2) gate as an example, where A, B and O are the inputs and output of AND-2; A_t, B_t and O_t are their security labels, respectively.

At the top level, we have the most precise hardware IFT logic for AND-2. The second level shows three simplified versions of hardware IFT logic derived by setting one of the inputs to don't-care. By eliminating an additional input, we reach the least precise (but most simplified) hardware IFT logic for AND-2 at the bottom.

$$\begin{tikzcd}
 & \mathrm{Ot = ABt + BAt + AtBt} \arrow[dl, "\mathrm{Ot = \cancel{A}Bt + BAt + AtBt}"'] \arrow[dr, "\mathrm{Ot = ABt + \cancel{B}At + AtBt}"] & \\
\mathrm{Ot = Bt + BAt} \arrow[dr, "\mathrm{Ot = Bt + \cancel{B}At}"'] & & \mathrm{Ot = ABt + At} \arrow[dl, "\mathrm{Ot = \cancel{A}Bt + At}"] \\
 & \mathrm{Ot = At + Bt} &
\end{tikzcd}$$

Fig. 7. Deriving simplified hardware IFT logic for AND-2.

In [52], both graph and QBF-SAT based search techniques are proposed to gradually introduce false positives into hardware IFT logic to accelerate security verification and pinpoint security vulnerabilities. Becker *et al.* took a step further to quantitatively introduce a certain percentage of false positives into IFT logic with the assistance of QBF solvers, allowing a finer-grained tradeoff between precision and complexity [7].

4 SECURITY PROPERTIES

Information flow properties are expressed as flow relations over objects. The property specifies how information stored in hardware objects can or can not flow to other objects. For example, a confidentiality property is specified by stating certain flows cannot occur (e.g., `secret` $\nrightarrow$ `unclassified`). An integrity property is also specified by restricting flows (e.g., `untrusted` $\nrightarrow$ `trusted`).

IFT properties are examples of *hyperproperties* since they express expected behaviors that require verification across multiple traces [22]. A *trace* is an execution of the hardware, e.g., a set of waveforms denoting the values of the hardware registers over time. An individual trace can be determined to satisfy or violate a *trace property*. Functional hardware verification techniques use trace properties, which are defined over sets of traces, e.g., a register should be set to ‘0’ upon reset. While trace properties can encode functional properties, IFT properties require comparing the hardware’s behavior with respect to more than one trace. Hyperproperties are defined on sets of traces (also called systems) and thus are more complex than trace properties.

To illustrate the relationship between hyperproperties and hardware IFT, consider an *integrity* property, which indicates that the behavior of trusted objects should remain intact with respect to values of untrusted objects. This policy compares the hardware behavior over multiple traces with differing values for untrusted data. Consequently, the integrity policy cannot be represented as a trace property. However, integrity can be represented as a hyperproperty that considers multiple traces which are identical except for the value of the untrusted data. To preserve integrity, the system is expected to act identically (in terms of the trusted objects) across all the traces assuming they start from equivalent initial states. Using formal methods, this hyperproperty can be verified for all possible values of untrusted data components to provide a formal proof of integrity in all possible executions. One could also verify the hyperproperty through simulation or emulation by ensuring that the security label never reach an undesired state. This likely will not provide total coverage, but typically scales better than formal verification techniques.

Hardware IFT tools model the movement of information through the hardware using labels that store security-relevant information. The labels indicate whether objects are trusted or untrusted, confidential or public, contains timing variation or not, etc. The property defines the initial conditions on the labels and a set of conditions for how the labels can or can not change during execution. For an integrity property, IFT uses labels to indicate if objects are trusted or not. Objects which carry untrusted values are initialized to have a *high* label while other objects have a *low* label by default. To ensure integrity, the property states that the system compute in a manner that maintains the *low* labels for the objects which are assumed to be trusted throughout execution. Maintaining a *low* label in this scenario demonstrates that untrusted values have not influenced the trusted components.

In this section, we show how different security policies can be modeled from the perspective of information flow and provide example hardware IFT properties. We describe the properties as an assertion-based IFT property language [49]. Assertion-based verification is commonly used for functional verification [41], and thus should be familiar to hardware designers. Additionally, commercial hardware design tools have adopted assertion-based verification for IFT properties [97, 98]. Assertion-based IFT verification works by initially setting labels in the system (through *assume* statements) and verifying that these labels do or do not propagate to some other object (through *assert* statements). We consider an IFT model where relevant hardware components are extended with a security label, and a two level lattice `low` $\sqsubseteq$ `high` shown in Fig. 2 (a). We use the System Verilog Assertion (SVA) language to specify these properties.

4.1 Confidentiality

Confidentiality properties ensure that any information related to sensitive objects (labeled as `high`) never flows to an unclassified object (labeled as `low`). For instance, in a cryptographic core the secret key should not flow to a point that is publicly observable. Confidentiality properties are written to protect secret assets in a given design. This is done by marking the secret assets or inputs with a `high` security label and monitoring the label of public ports and storage units. To preserve confidentiality, all public ports and storage units should maintain a `low` label throughout execution. Examples of confidentiality properties for various hardware designs such as crypto cores and arithmetic units are shown in the first four rows of Table 1. Here we wish to verify that sensitive/secret objects do not affect publicly viewable objects. To do this, we set the sensitive objects labels as `high` using the `assume` statement, and `assert` that the publicly viewable objects remain `low`.

Table 1. Examples of hardware IFT properties related to confidentiality, integrity, and isolation.

Benchmark	Synopsis	Formal Representation
SoC Arbiter	(Confidentiality) Acknowledgement signal is not driven from sensitive requests	`assume(req[i] == high);` `assert(ack[j] == low)`
Scheduler	(Confidentiality) Grant signal is not driven from sensitive modes	`assume(mode[i] == high);` `assert(grant[j] == low)`
Crypto Core	(Confidentiality) Ready signal is not driven from secret inputs	`assume(key == high & plain_text == high);` `assert(ready == low)`
Floating Point Unit	(Confidentiality) Ready signal is not driven from the inputs	`assume(operand1 == high & operand2 == high);` `assert(ready == low)`
Crypto Core	(Integrity) Key register is not modified by public inputs	`assume(user_inp == high);` `assert(key == low)`
Debug Unit	(Integrity) Debug flag is not modified by public inputs	`assume(user_inp == high);` `assert(debug_en == low)`
Processor	(Integrity) PC, private memory and control flow conditions are not modified by public inputs	`assume(user_inp == high); assert(PC == low)` `assume(user_inp == high); assert(private_mem == low)` `assume(user_inp == high); assert(cond == low)`
Access Control	(Integrity) Unauthorized users cannot access protected units	`assume(user[i] == high);` `assert(asset[j] == low)`
SoC	(Isolation) Accesses to different cores on an SoC are isolated	`assume(req[i] == high); assert(ack[j] == low)` `assume(req[j] == high); assert(ack[i] == low)`
Memory	(Isolation) Memory locations are isolated	`assume(mem[i] == high); assert(mem[j] == low)` `assume(mem[j] == high); assert(mem[i] == low)`

4.2 Integrity

Integrity is the dual of confidentiality - here we mark untrusted hardware resources with a `high` label and verify that they do not affect critical components with `low` labels. For example, in a processor the program counter (PC) should not be overwritten by data from an unprotected network.

Integrity properties can be specified for any design where certain memory locations, registers, or flags should be protected against unauthorized access. This is modeled by marking public access such as user or network input with a `high` label and constraining the sensitive variables to maintain a `low` security label. Table 1 shows four integrity properties written for crypto cores, debug units, processors, and access control units.

4.3 Isolation

Isolation can also be enforced as an information flow security property. Isolation states that there should never be information exchange between two components with different trust levels. For example, in SoC designs, trusted IP cores sitting in the secure world with `low` labels should be separated from those which are untrusted and are in the insecure domain with `high` labels. It should be noted that isolation is a two-way property as shown in the examples of Table 1.

4.4 Constant Time

Information flow models can be used to capture timing side-channels resulting from runtime variations in hardware designs. Constant time properties assess whether sensitive information can be retrieved by measuring the computation time. To precisely capture timing flows, the information flow model needs to distinguish between logical and timing flows. Take as example a floating point division unit expected to run in constant time independent of the value of the operands. In this case, logical flow exists from the data inputs (i.e., the divider and dividend) to the data outputs (i.e., quotient and remainder) since the outputs are computed from the inputs. However, whether or not there is timing flow from the inputs to the outputs (i.e., if the arithmetic unit runs in constant time or not) depends on the implementation of the floating point unit.

Table 2 summarizes the properties used in [2, 4] to verify timing side channel in different hardware designs. Here, we assume an IFT model where the flow relation tracks timing-based information flows (denoted by *time*), e.g. as described in Clepsydra [2]. To verify timing leakage, the assertions are written over these "*time*" labels.

Table 2. Summary of properties used for detecting timing side channes.

Benchmark	Synopsis	Formal Representation
Sequential Divider	Result is ready in constant time	`assume(dividend == high & divisor == high);` `assert(quotient` $==_{time}$ `low & remainder` $==_{time}$ `low)`
Sequential Multiplier	Result is ready in constant time	`assume(dividend == high & divisor == high);` `assert(quotient` $==_{time}$ `low & remainder` $==_{time}$ `low)`
Cache	Data is available in constant time	`assume(index[i] == high ); assert(data[j]` $==_{time}$ `low)`
SoC Arbiter	Requests are granted in constant time	`assume(req[i] == high ); assert(ack[j]` $==_{time}$ `low)`
Thread Scheduler	Scheduling is done in constant time	`assume(thread_active[i] == high );` `assert(thread_grant[j]` $==_{time}$ `low)`
AES Cipher	Cipher text is ready in constant time	`assume(key == high & plain_text == high);` `assert(cipher_text` $==_{time}$ `low)`
RSA Cipher	Cipher text is ready in constant time	`assume(key == high & plain_text == high);` `assert(cipher_text` $==_{time}$ `low)`

4.5 Design Integrity

IFT can be used to detect certain types of undocumented design modifications (known as hardware Trojans) that leak sensitive information or overwrite critical memory locations by inserting malicious information channels in the design. For example, Trust-HUB benchmarks [129] include examples of Trojans added to crypto cores using a small circuitry that transfers the secret key to a public output under certain conditions. This class of hardware Trojans can be detected using a property that observes information flow from sensitive data (e.g., the secret key) to the public ports. Table 3 summarizes the properties for detecting information leakage by GLIFT in the Trust-HUB [129] benchmarks as used by Hu et. al. [53].

Table 3. Summary of properties used for detecting hardware Trojans

Benchmark	Synopsis	Formal Representation
AES-T100	Key does not flow to Antena	`assume(key == high);` `assert(Antena == low)`
AES-T400	Key does not flow to shift register	`assume(key == high);` `assert(TSC-SHIFTReg == low)`
AES-T1100	Key does not flow to capacitance	`assume(key == high);` `assert(capacitance == low)`
RSA-T200	Key does not flow to count	`assume(key == high);` `assert(count == low)`

4.6 Common Weakness Enumeration

The Common Weakness Enumeration (CWE) is a community-developed list of software and hardware weakness types [96]. The hardware CWE list started in 2019 and includes a large number of enumerations that can be modeled using IFT properties. This includes most of the CWE categories including CWE-1195: Manufacturing and Life Cycle Management Concerns, CWE-1196: Security Flow Issues, CWE-1197: Integration Issues, CSE-1198: Privilege Separation and Access Control Issues, CWE-1199: General Circuit and Logic Concerns, CWE-1201: Core and Compute Issues, CWE-1202: Memory and Storage Issues, CWE-1205 Security Primitives and Cryptography Issues, CWE-1207: Debug and Test Problems, and CWE-1208: Cross-Cutting Problems. Many of these weaknesses can be modeled as IFT properties discussed in this section. For example, Tortuga Logic claims their IFT-based security verification tool *Radix* [97, 98] can support over 80% of the hardware CWEs [99].

5 LEVEL OF ABSTRACTION

The modern hardware design process relies upon abstraction and electronic design automation (EDA) tools that translate system level specifications into detailed transistor level circuits. In the early stages of hardware development, designers focus on functionality and system level integration – ignoring lower level details like cycle accurate timing, logical implementation, and physical layout. System level specifications model computation as transactions and use electronic system level (ESL) synthesis tools to transform the transactions into untimed algorithmic descriptions. High level synthesis (HLS) tools compile algorithmic behaviors into architectural description. RTL is translated to the gate level by logic synthesis tools and to transistor circuit layouts using physical synthesis tools. Each level brings more detail, and thus more complexity.

Detecting potential security vulnerabilities earlier in the design process allows the designers to fix them at much lower cost and effort. The drawback is that designers need to make conservative assumptions about the hardware behaviors due to the lack of implementation details. The architectural level has enough design information for verifying the security related interference between hardware and software. The challenge lies in accurately accounting for explicit flows and timing channels at this level. As we move to RTL, the cycle accurate nature of the model of computation can capture timing information flows. However, the complex syntax features (e.g., conditional branch statements) complicates IFT analysis. The gate level models all information flows explicitly, which significantly lowers the complexity of the IFT analysis. Unfortunately, gate level simulation and verification is typically one or two orders of magnitude slower than doing the same job at the RTL [8].

Each abstraction level has different capabilities and accuracies in describing design behaviors, which in turn allow verifying of different types of security properties [85]. In this section, we use hardware abstractions to categorize and summarize IFT analysis techniques. We start at the system level and move through algorithmic, architectural, RTL, gate, and circuit level. Figure 8 describes the hardware design abstractions, a sampling of the IFT tools at these levels of abstraction, and provides an outline for this section.

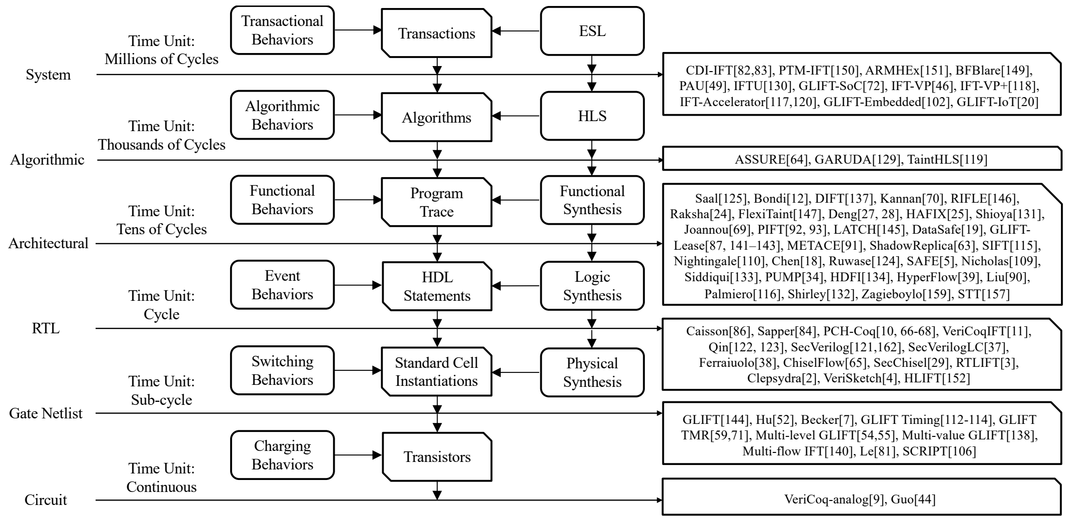

Transactional Behaviors
Transactions
ESL
Time Unit: Millions of Cycles
System
CDI-IFT[82,83], PTM-IFT[150], ARMHEx[151], BFBlare[149], PAU[49], IFTU[130], GLIFT-SoC[72], IFT-VP[46], IFT-VP+[118], IFT-Accelerator[117,120], GLIFT-Embedded[102], GLIFT-IoT[20]
Algorithmic Behaviors
Algorithms
HLS
Time Unit: Thousands of Cycles
Algorithmic
ASSURE[64], GARUDA[129], TaintHLS[119]
Functional Behaviors
Program Trace
Functional Synthesis
Time Unit: Tens of Cycles
Architectural
Saal[125], Bondi[12], DIFT[137], Kannan[70], RIFLE[146], Raksha[24], FlexiTaint[147], Deng[27, 28], HAFIX[25], Shioya[131], Joannou[69], PIFT[92, 93], LATCH[145], DataSafe[19], GLIFT-Lease[87, 141–143], METACE[91], ShadowReplica[63], SIFT[115], Nightingale[110], Chen[18], Ruwase[124], SAFE[5], Nicholas[109], Siddiqui[133], PUMP[34], HDFI[134], HyperFlow[39], Liu[90], Palmiero[116], Shirley[132], Zagieboylo[159], STT[157]
Event Behaviors
HDL Statements
Logic Synthesis
Time Unit: Cycle
RTL
Caisson[86], Sapper[84], PCH-Coq[10, 66-68], VeriCoqIFT[11], Qin[122, 123], SecVerilog[121,162], SecVerilogLC[37], Ferraiuolo[38], ChiselFlow[65], SecChisel[29], RTLIFT[3], Clepsydra[2], VeriSketch[4], HLIFT[152]
Switching Behaviors
Standard Cell Instantiations
Physical Synthesis
Time Unit: Sub-cycle
Gate Netlist
GLIFT[144], Hu[52], Becker[7], GLIFT Timing[112-114], GLIFT TMR[59,71], Multi-level GLIFT[54,55], Multi-value GLIFT[138], Multi-flow IFT[140], Le[81], SCRIPT[106]
Charging Behaviors
Transistors
Time Unit: Continuous
Circuit
VeriCoq-analog[9], Guo[44]

Fig. 8. Hardware IFT techniques exist at all levels of the hardware design.

5.1 System Level

A number of approaches explore hardware extension for dynamic information flow tracking (DIFT) to perform system level verification [21, 48, 72, 82, 83, 118, 132, 151, 153]. Some techniques use loosely-coupled processor or dedicated hardware to perform IFT at runtime while others employ IFT to verify system security during design time.

5.1.1 IFT Through Co-processing Unit. Lee *et al.* use CDI (Core Debug Interface) to fetch meta data [82, 83]. The ARM CoreSight ETM (Event Trace Macrocell) trace component is used to collect information for each CPU instruction. This trace component is later updated to CoreSight PTM (Program Trace Macrocell), which provides information only on instructions that modify the PC. Wahab *et al.* use this new debug feature in ARMv7 processor to retrieve details on instructions committed by the CPU to perform DIFT. A DIFT coprocessor is implemented in the program logic of Xilinx Zynq device [152] for tag processing. *ARMHEx* [153] is a DIFT solution leveraging the CoreSight PTM debug component. Static analysis and instrumentation are employed to retrieve the missing information resulting from the update. ARMHEx also considers the security of the DIFT unit by using ARM Trust-Zone to protect it. A successive work develops a more adaptive information flow monitor called *BFBlare*, which supports multi-thread, virtual tag memory, floating point and Linux kernel features [151].

Heo *et al.* propose an application specific instruction set processor (ASIP) architecture for system level programmable dynamic program analysis [48]. The proposed approach is implemented as a hardware engine called program analysis unit (*PAU*), whose key components are the tag processing core and the main controller. The main controller communicates the execution traces with the host processor through the system bus. The tag processing core is an application specific instruction set architecture for management of tags.

Shin *et al.* propose the implicit flow tracking unit (*IFTU*), an external hardware module connected to the host processor through system bus. In IFTU, for a conditional branch, the taint tag of data used for branch condition checking is first propagated to the program counter. Then, for the instructions after the conditional branch, the taint tag of the program counter is propagated to the destinations where the values are affected by the branch result. IFTU can detect more recent advanced attacks that exploited implicit flows at reasonable area costs [132].

5.1.2 IFT for Verifying and Enforcing SoC Security. Kastner *et al.* use IFT to prevent secret information from leaking (for confidentiality), and untrusted data from being used to make critical decisions (for integrity) in SoC systems using untrusted IPs [72]. They use a gate level information flow model but use it to verify system level properties.

Hassan *et al.* and Pieper *et al.* propose an approach for SoC security validation through Virtual Prototypes (VP) modeled using SystemC TLM 2.0 [47, 119]. They employ static information flow analysis to verify the *no flow* security property and detect potential security breaches such as data leakage and untrusted access. However, information flow is determined by static path search between the source and sink points, which can lead to conservative verification results.

Piccolboni *et al.* add support of DIFT to loosely coupled accelerators in heterogeneous SoCs, which are vulnerable to attacks [118]. Their work is inspired by two earlier works [120, 121] on DIFT targeting such accelerators. A standalone shell circuit is added for interacting with the accelerator, e.g., loading, propagating and storing tags.

5.1.3 IFT for Enforcing Embedded System Security. In [103], a bottom-up approach is proposed for enforcing information flow security in embedded systems. The paper implements bit-tight information flow control from the level of Boolean gates. It then exposes this information flow measurement capability to higher abstraction levels in the system stack, ranging from architecture all the way up to secure applications to enforce security properties.

Cherupalli *et al.* propose a fine-grained IFT technique for enforcing information flow security for IoT systems [21]. They introduce application specific GLIFT [146] for IoT systems and build a gate-level symbolic analysis framework that leverages knowledge of the target application in order to efficiently identify potential information flow security vulnerabilities. They design a software tool that takes as inputs the gate level netlist of the processor, the original binary of a given application and information flow policies of interest. The tool performs symbolic gate level simulation of the application binary on the netlist to determine if there is any information flow security policy violation, and if so to identify the offending instructions.

5.2 Algorithmic Level

Some recent works synthesize DIFT enhanced hardware accelerators from high-level specification [64, 120, 131].

ASSURE [64] is a security-constrained high-level synthesis (HLS) framework that automatically synthesizes verifiably secure hardware accelerators from high-level descriptions under user-defined security policies and constraints. ASSURE extends the LegUp HLS framework [18] to perform IFT. It generates an additional enforcement FSM to manage public I/O accesses separately in a way that all secret-conditioned branches are balanced by applying additional latency constraints. Other operations that do not affect timing of the public outputs are controlled by the main FSM. As a result, the public outputs have constant latency.

GARUDA [131] is a high-level security policy language and a compiler to compile security policies described in GARUDA to Verilog modules. It enables the modular construction and composition of runtime hardware monitors for securing a variety of security policies, including fault isolation, secure control flow, and DIFT via taint checking. The high-level security policies described in GARUDA are first converted into a policy intermediate representation (IR) and then compiled to Verilog code using the extraction feature of the Coq theorem prover.

TaintHLS [120] is an HLS methodology to automatically generate DIFT-enabled accelerators from high-level specification. TaintHLS creates a microarchitecture to support baseline operations and a shadow microarchitecture for intrinsic DIFT support in hardware accelerators. The DIFT-enabled accelerators have negligible performance overhead while adding no more than 30% hardware. These accelerators achieve the same value tags and identical number of false positives/negatives as compared to software IFT implementation.

5.3 Architecture Level

5.3.1 Capability Vector Architecture. Architectural level hardware IFT research dates back to as early as Denning's information flow theoretic framework. In 1978, Saal and Gat from IBM proposed a *capability vector machine* to enforce access and information flow control [127]. The machine used a capability vector to describe information flow policy. For example, an instruction moving data from A to B required the capability vectors <(A, *READ*) and (B, $WRITE$)>. An instruction would be allowed to execute when its implied capability vector existed; otherwise, the movement of information should be prohibited.

Bondi and Branstad designed an architecture for fine-grained secure computing [13]. The architecture's operands are 2-tuples, consisting of a data object and an associated security tag. A dedicated subprocessor and tag memory are added to perform security tag manipulation in parallel to the data path. The coupled subprocessors enforce MLS access control and information flow control. While such an architecture provides efficient security support, it is infeasible for modern processors with their large instruction sets.

5.3.2 DIFT and Related Architectures. *DIFT* is an architectural mechanism to protect programs against buffer overflow and format string attacks by identifying spurious information flows from untrusted I/O and restricting the usage of untrusted information [139]. Although DIFT employs conservative tracking rules, the performance overheads can be as high as 23% for certain benchmarks. To reduce overhead, Kannan *et al.* propose to decouple DIFT with a co-processor [70]. The co-processor receives instruction tuples and loads tag information from tag memory/cache to perform security checks. The two processors synchronize on system calls. When a tag check fails, an exception is generated and the execution results of the instruction and tag update are rejected.

Runtime Information Flow Engine (*RIFLE*) enforces user-defined information flow security policies [148]. In RIFLE, program binaries are translated from a conventional ISA to an information flow secure ISA. The translated programs execute on hardware with IFT capability and interact with a security-enhanced operating system, which is responsible for enforcing the security policies. The authors also show that language-based information-flow systems are less secure than previously thought while architectural level IFT approaches can be at least as powerful as language-based information-flow systems.

Raksha is an architecture that combines both hardware and software DIFT mechanisms [25]. The hardware manages security tags at low performance overhead for user code, OS code, and data that crosses multiple processes, while the software performs security checks and handles exceptions. Raksha supports a flexible and programmable mechanism for specifying security policies, allowing multiple concurrently active security policies to deal with a wide range of attacks. It enables security exceptions that run at the same privilege level and address space as the protected program. This allows the integration of the hardware security mechanisms with software analyses, without incurring the performance overhead of switching to the operating system.

FlexiTaint [149] is another hardware accelerator for DIFT with support of flexible and configurable taint propagation rules. It extends the processor pipeline and implements taint-related logic for tag processing. Tags are stored as a packed array in virtual memory; a taint propagation cache tagged by operation type and input-operand taints to improve performance. Reported results show that FlexiTaint incurs negligible performance overheads even when simultaneously supporting two different taint propagation policies.

In [28, 29], Deng *et al.* implement DIFT using a soft-core processor. The CPU pipeline is modified in order to allow the processing core to forward its execution trace for tag processing. In [26], executable binaries are instrumented to extract the required information for DIFT. However, this incurs up to 86% performance overhead.

Shioya *et al.* and Joannou *et al.* tried to reduce the memory overhead of DIFT techniques by exploiting tag memory access characteristics, e.g., non-uniformity, locality, and cache behavior. They propose the use of multi-level table, cache structures and memory access optimizations to accelerate tag memory access [69, 133]. Another attempt to optimize taint label manipulation is to separate `trusted` and `untrusted` data objects into different pages, which prevents the overhead for storing and loading taint labels [92, 93].

Locality-Aware Taint CHecker (*LATCH*) is an architecture for optimizing DIFT [147]. The key observation is that DIFT exhibits strong temporal locality, with typical applications manipulating sensitive data during limited phases of computation. This property allows LATCH to invoke precise, computationally intensive tracking logic only during execution involving sensitive data and otherwise performing lightweight, coarse-grained checks, which reduces performance overheads. The authors design and implement three different DIFT systems that incorporate the LATCH model, i.e., S-LATCH for optimizing software-based DIFT on a single core, P-LATCH to optimize DIFT monitoring using a separate core and H-LATCH that targets hardware-based DIFT.

DataSafe is a software-hardware architecture that provides dynamic instantiations of secure data compartments (SDCs) [20]. The architecture enforces hardware monitoring of information flows from compartments using hardware policy tags associated with the data at runtime in order to prevent the leakage of sensitive information. Security tags are derived from security policies associated with sensitive data and passed to the SDC for tag propagation. These tags are checked when data is processed by unvetted applications.

5.3.3 Execution Lease Architectures. Tiwari *et al.* describe how a class of secure architectures can be constructed, from the gates up, to completely capture all information flows [143–145]. In [144], hardware IFT is applied to an FPGA device to create a micro-processor that implements bit-tight information flow control. The architecture design is a full implementation that is programmable and precise enough to track all flows of information and prevent untrusted data from flowing to critical memory locations such as the PC. Novel techniques are proposed to handle conditional branches, loops, loads and stores to prevent over tainting. A prototype processor is implemented on Altera Stratix II FPGA, where IFT logic adds about 70% resource overhead [145].

The *execution lease* is another IFT-enhanced architecture [87, 143]. This lease architecture allows execution contexts to be tightly quarantined and their side effects to be tightly bounded. Information flow security properties such as isolation between trusted and untrusted execution contexts can be verified all the way down to the gate-level using the precise information flow measurement capability of IFT. However, all the leases have a fixed time bound, which does not support performance optimization micro-architecture features such as caches, pipelining, branch prediction or TLBs due to the timing variation they introduce. In addition, the lease architecture adds substantial design and performance overheads resulting from the additional IFT logic.

To overcome the shortcomings of the lease architecture, in [145] a minimal but configurable architectural skeleton is crafted to operate alongside a small piece of software, which together composes the minimal essential functionality through which information flow security properties can be verified all the way down to Boolean gates. This enhanced core component is then used to create a hardware-software system that allows unbounded operation, inter-process communication, pipelining, I/O with traditional devices and other architectural-level performance optimizations. This work also proposes a more scalable verification technique called star-Logic (*-logic), which represents indeterministic values as the abstract value * in order to verify all possible executions arising from unknown values in a single run.

5.3.4 IFT on Multi-core Architectures. Most architectural level IFT techniques discussed above use a dedicated co-processor for tag processing. There are also works that distribute data processing and metadata processing to different cores in a single processor [19, 63, 91, 106, 111, 116, 126]. Implementing DIFT on multi-core processors can be a much more challenging task, since data processing and metadata processing are normally decoupled and in different orders. Maintaining metadata coherence is a fundamental problem for ensuring the correctness of DIFT in a multi-core environment. To address possible inconsistency, Santos *et al.* proposed *METACE*, which includes architectural enhancement in the memory management unit and leverages cache coherence hardware protocol to enforce metadata coherence [91]. In addition, there is also performance penalty due to inter-process communication [106].

ShadowReplica [63] is another dynamic data flow tracking technique on multi-core processors. It uses a shadow thread and spare CPU cores in order to decouple execution and data flow tracking. The two processes communicate through a shared data structure. ShadowReplica differs from previous approaches in that it introduces an off-line application analysis phase that utilizes both static and dynamic analysis methodologies to generate optimized code for decoupling execution and implementing DFT. This additional phase also minimizes the amount of information that is to be communicated between the two threads.

SMT-based IFT (*SIFT*) is a hardware IFT technique on SMT processors [116]. Taint propagation and policy checking are performed by a separate thread executed in a spare context of an SMT processor. The instructions for the checking thread are generated in hardware using self-contained off-the-critical path logic at the commit stage of the pipeline. Instruction generation can be completed in one additional cycle at commit time. Experimental results using SPEC CPU 2006 benchmarks showed 4.5% area overhead, 20% overhead in performance and 23% additional power consumption, which is significantly lower than several software IFT implementations.

Another benefit of performing IFT on multi-core processors is the possibility of parallelizing DIFT in order to reduce the high performance overhead of sequential implementations. The major challenge lies in serial dependencies in DIFT, i.e., label propagation usually depends on the resulting labels of prior operations. To accelerate DIFT, different optimizations are proposed to resolve and simplify such dependencies, e.g., preserving only operations that affect the taint status of critical points [111] and tracking the information flow only through unary operations [19, 126]. Afterwards, the parallelized DIFT instances are distributed to multiple cores for processing.

5.3.5 IFT on RISC-V Architectures. The *SAFE* project developed one of the earliest verified secure information flow architectures based on the RISC instruction set. A simple stack-and-pointer machine with “hard-wired” dynamic IFC was designed to provide IFT support to higher-level software. The instructions for implementing the IFT mechanism are formalized and verified using Coq [5]. More recent projects such as *SSITH* and *AISS* both aim to develop secure RISC-V architectures, where IFT is employed for verifying and enforcing security guarantees [110, 135].

Programmable Unit for Metadata Processing (*PUMP*) is an enhanced RISC processor architecture with ISA-level extension [35]. PUMP changes the pipeline to incorporate programmable tag propagation rule checking. PUMP performs single-cycle common-case computation on metadata with support of the tag propagation rule cache. On every instruction, the tags of the inputs are used to determine if the operation is allowed, and if so to determine the tags for the results. A tag propagation rule cache miss will trigger the policy miss handler to determine if the instruction is allowed.

HDFI is a hardware-assisted solution for data flow isolation by extending memory words with single bit tags and the RISC-V ISA to perform tag checking [136]. The integrity level of a memory unit is determined by the last write operation to the unit. At memory read, a program would check if the tag is allowed by the security policy. HDFI enforces the Biba integrity and Bell-LaPadula confidentiality security models with $< 2\%$ performance overhead.

HyperFlow [40] is another extended RISC-V architecture with security features for enforcing information flow control. The processor is designed and verified using a type enforced hardware design language *ChiselFlow* [65]. It supports rich configurable security policies described by complex lattice structures and controlled security class downgrading to allow inter-process communication and system calls. This new architecture also prevents timing channels by enforcing the security policy that the latency of an instruction should not depend on operand value or memory access from another process with a higher security type. A later work extends this architecture by also incorporating language-level IFC for securing the system architecture of secure autonomous vehicles [90].

Several other works build hardware IFT capability into RISC-V processors in order to provide hardware-assisted security [117, 134]. In [160], an ISA based on IFC is designed to mitigate the gap between IFC hardware and software. This ISA is used to prove strong timing-sensitive security conditions about software, leveraging the IFC capabilities and guarantees provided by hardware.

5.3.6 IFT for Speculative Architecture. Speculative Taint Tracking (*STT*) [158] is a framework that executes and selectively forwards the results of speculative access instructions to younger instructions, as long as those younger instructions cannot form a covert channel. STT tracks the flow of results from access instructions in a manner similar to DIFT, until those results reach an instruction, or sequence of instructions, that may form a covert channel. It automatically “untaints” the result once the instruction that produces it becomes non-speculative, in order to improve performance. The framework enforces a novel form of non-interference, with respect to all speculatively accessed data.

5.4 RTL

A large body of research work aims to develop hardware IFT techniques at the RTL level or extend existing HDLs with IFT capability [2–4, 10, 11, 30, 38, 66–68, 84, 86, 163].

5.4.1 State Machine Based IFT Language. *Caisson* [86] and Sapper [84] are HDLs enhanced with IFT capability for the automated generation of circuits that enforce IFT-related security properties. Both languages model hardware designs using state machines and prevent illegal flows from a `high` state to a `low` state. In Caisson, security policies are enforced through static type checking. However, this technique can cause large area overheads since static property checking requires that resources be hard partitioned or even replicated. *Sapper* improves Caisson by taking a hybrid approach to label checking. It uses static analysis to generate a set of dynamic checks that are inserted into the hardware design in order to enable dynamical tracking. This enables resource re-use and thus lowers the area overheads. In addition, Sapper also made an attempt to enforce timing-sensitive noninterference by accounting for information flows related to when events happen.

5.4.2 PCH and Coq Based IFT. Makris and Jin *et al.* integrated IFT into the proof carrying hardware (*PCH*) framework [10, 11, 66–68]. They define Verilog-to-Coq conversion rules [11] to automate Verilog design to Coq formal logic translation. The resulting Coq semantic circuit model facilitates tracking secrecy labels and proving information flow security properties. To track the flow of information, the Coq circuit model is extended with a sensitivity label so that the Coq circuit primitives return a `value*sensitivity` pair, where `sensitivity` can be either `secure` or `normal`. The enhanced circuit model is then formally verified under the Coq proof environment to check if the design adheres to desired security properties. A security violation would be identified when a `normal` signal takes on a `secure` tag.

The PCH framework provides a method for protecting the hardware design and supply chain. However, it employs a conservative sensitivity label propagation policy, i.e., always allowing `secure` labels to propagate independent of the operation performed or the input values. While this reduces the complexity of the formal Coq circuit model and allows fast profiling of security properties, conservative IFT usually leads to false positives (as we discuss in Section 3). Qin *et al.* made an attempt to eliminate such false positives by incorporating the precise security label propagation policy from GLIFT [146] into the PCH framework [123, 124]. This allows GLIFT logic to be represented in Coq syntax and verified using a new proof tool, but it incurs large overheads in the complexity of the Coq circuit model and proof time.

5.4.3 SecVerilog and SecChisel. *SecVerilog* is a language-based approach for enforcing information flow security [163]. SecVerilog augments Verilog with labels and performs static type checking to determine if the design meets the desired security properties. It enforces timing-sensitive noninterference and can be used to design secure architectures that are free of timing channels. The research group later improves the precision of SecVerilog by using finer-grained labels to reason about the security of individual array memory elements and bits in data packets. The improved security-type based HDL can be used to verify security properties such as noninterference, and can uncover hardware vulnerabilities [39]. SecVerilogLC supports mutable dependent types to allow secure sharing of hardware resources. A major challenge addressed is implicit downgrading [163] caused by unsynchronized updates of data values and security labels. SecVerilogLC proposes a new type system and introduces an explicit notion of cycle-by-cycle transitions into the syntax, semantics, and type system of the new language to solve the implicit downgrading problem [38].

ChiselFlow is another type-enforced HDL with timing labels for developing secure hardware architectures [40]. It has been used to create cryptographic accelerators with hardware-level information flow control and formally verify the security of these implementations [65]. A successive work extends the Chisel hardware design language with security features to create a design-time security verification framework, which employs information flow analysis to verify hardware security properties in order to capture information leaks caused by hardware security flaws and Trojans [30].

5.4.4 RTLIFT. *RTLIFT* precisely captures all logical flows, including timing flows, leveraging the cycle-accurate timing information available at the RTL level [3]. The major challenge lies in precisely accounting for the implicit flows resulting from complex RTL syntax such as nested conditional branches. This is resolved by converting the branch statements to multiplexer network so that one only needs to instantiate IFT logic for the multiplexers. RTLIFT allows flexible precision and complexity tradeoffs by instantiating different versions of tracking logic for logical and arithmetical primitives. It achieves over 5X speedup in verification performance over GLIFT [146] and thus has better scalability.

Clepsydra [2] provides a new IFT model for distinguishing timing flows from functional ones, and formally verifying timing channel security. A timing information flow occurs when a register is conditionally updated under the control of a tainted value. In addition to the taint label in IFT, Clepsydra introduces a timing label to track the propagation of timing variation. The Clepsydra tool can formally prove either the existence of a timing leakage or constant execution time. The *VeriSketch* [4] tool takes it a step further and automatically synthesizes hardware designs that satisfy information flow security properties, using the sketch technique for synthesizing designs that meet specified functional constraints.

5.4.5 Control and Data Flow Graph Based RTL IFT. High-level information flow tracking (*HLIFT*) was developed for detecting hardware Trojans that leak secret information through exclusive output [154]. This method extracts IFT features from coarse grained control and data flow graph (CDFG) of hardware Trojan benchmarks. Trojan detection is performed by matching the statement level CDFG of given RTL with the extracted feature library.

5.5 Gate Level

Researchers also aimed to understand the flow of information through Boolean gates. This can be a complement to higher-level IFT methods when the hardware design comes in the form of gate netlist, e.g., an IP core. In addition, the complex syntax structures such as conditional branch statements are all flattened to primitive gates and thus, hardware IFT model can be created with significantly lower effort compared to higher levels of abstraction.

5.5.1 GLIFT. GLIFT is a well-established IFT technique [146]. A key insight is that all information flows, whether explicit or implicit, all appear in a unified form at the gate level and have good mathematical representations. GLIFT employs fine-grained taint label and label propagation policy to account for each bit of information flow. It precisely measures the actual flows of information by analyzing the influence of inputs on the output and greatly eliminates the false positives of conservative IFT techniques. As an example, GLIFT precisely indicates that a `low` *reset* clears the `high` label of a counter while conservative IFT methods cannot.

Hu *et al.* built the theoretical fundamentals of GLIFT [57, 112]. They derived formal representations of GLIFT logic for arbitrary primitive gates. A successive work [58] formally proved that precise GLIFT logic generation is an NP-complete problem. It then presented various algorithms for creating GLIFT logic for large digital circuits. The authors also identified the root cause of the differences in precision of GLIFT logic created using different methods as static logic hazards or variable correlation resulting from reconvergent fanouts. Such differences motivated IFT precision and complexity tradeoff research [7, 52] discussed in Section 3.4.

Several works attempt to reduce the complexity of GLIFT logic by employing optimized label encoding techniques [56, 59, 71]. The key observation is that the value of a tainted signal can be ignored in taint propagation, i.e., GLIFT logic will produce identical output taint label when a tainted bit takes the value of either `0` or `1` [59]. Thus, we can combine the tainted states and reduce the possible signal states to three, namely `untainted 0`, `untainted 1` and `tainted *`. The don't care condition in tainted signals provides flexibility for circuit optimization. Further, when `untainted 0`, `untainted 1` and `tainted *` are encoded as `00`, `11` and `01` (or `10`) respectively, the GLIFT logic can serve as hardware redundancy if we set all inputs to be `untainted`. This allows both information flow security and fault tolerance properties to be enforced with the same GLIFT logic [59, 71].

5.5.2 Multi-level, Multi-valued and Multi-flow GLIFT. In [54, 55], Hu *et al.* expand GLIFT for multilevel security lattices for enforcing MLS. They generalize the GLIFT method to arbitrary linear and non-linear security lattices by defining unified yet precise class combining operators. Another work expands GLIFT for multi-valued logic systems [140]. It derives taint propagation rules for additional design states such as *unknown* and *high impedance*, in order to support multi-valued IFT simulation. Multi-flow IFT [141, 142] is another recent extension to GLIFT. It aims to understand the effect of multiple inputs on a signal, e.g., how many key or random number bits are affecting a target bit at the same time. This technique enables modeling simultaneous information flow behaviors and proving quantitative information flow security properties.

5.5.3 GLIFT Enhanced with Timing Label. Oberg *et al.* demonstrated how GLIFT could be used to enforce timing information flow security [113–115]. In [113], GLIFT is employed for testing timing information flow security in shared bus architectures such as I^2C and USB. GLIFT shows that there can be unintended interference between trusted and untrusted devices through timing-related design behaviors. They use GLIFT to test for information flow isolation properties between trusted and untrusted IP components, which share computing resources such as the WISHBONE system bus and cryptography core [115]. A later work [114] formalizes the notion of timing channel and provides criteria to separate timing flows from functional ones. A *Timing-only flow* is defined as the case in which a set of inputs affects only the timing of the output. As an example, a change in the key of the RSA cipher affects when the encryption done signal is asserted. The paper then formally proves that GLIFTed state machines can detect timing-only flows.

5.5.4 Gate Level IFT Through Structural Checking. Another work by Le *et al.* tracked data flow in gate level design netlist through structural checking [81]. The proposed method assigns "asset" (or "property") to signals in hardware designs and then tracks the propagation of "asset" via data flow. It models "asset" related to cryptographic core, debug interface, timing channel and hardware Trojan. By observing the resulting "asset" of signals, the designer can understand important security properties such as integrity and confidentiality. However, structural checking is basically conservative connectivity analysis and thus cannot precisely determine if and when "asset" could propagate like GLIFT.

5.6 Circuit Level

Bidmeshki *et al.* made a first attempt to understand the flow of information in analog/mixed-signal designs [9]. Their work defines information flow policies for various analog components such as MOSFETs, bipolar transistors, capacitors, inductors, resistors and diodes, and extends the existing PCHIP-based IFT method [67, 68] established for digital designs to analog/mixed-signal circuits. While providing information flow control at the level of transistors is desirable due to recent security attacks that target analog hardware, IFT in analog/mixed-signal hardware designs can be a challenging problem. How to encode taint information using continuous analog signals, and precisely determine their influence on the output still needs to be investigated.

VeriCoq [11] is recently extended to track sensitive signals in the mixed-signal domain [45]. It detects charge-domain Trojans, a generalization of the A2 Trojan [157] and RowHammer [73] attacks. It proposes an abstracted model of charge-domain leakage structures tainted information flowing to user controllable or accessible flip-flops. Data flow graph based path search is employed for taint propagation along analog components such as MOSFET, Diode and Resistor. A charge-domain Trojan is identified if there is a path between the taint source and target.

Apart from a hierarchical view of hardware IFT methods at different levels of abstraction, a temporal view helps understand the developing trend in this realm. We mark a few milestones in the development of hardware IFT research as shown in Fig. 9. Early hardware IFT methods focused on the architectural level coupled by numerous works in language based IFT techniques [128]. The strong connection between instructions in software programs and functional units in computer architecture enables a natural extension of the IFT spectrum to the hardware domain. Gate-level IFT methods allow finer grained measurement and classification of information flow. More recent IFT developments

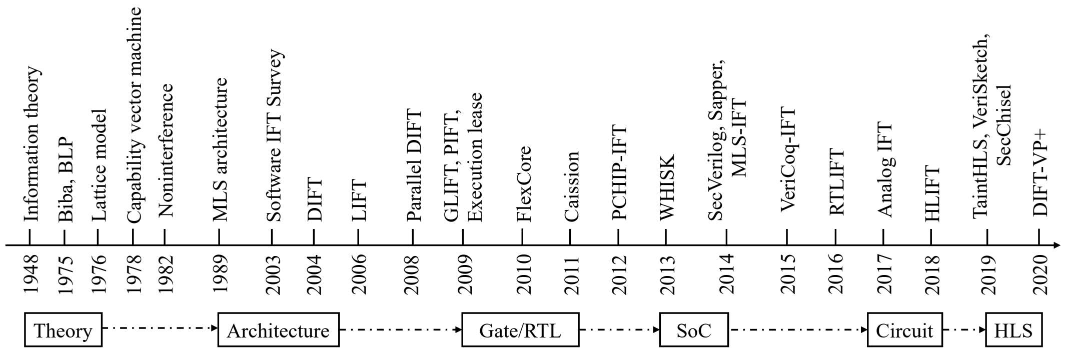


Fig. 9. Milestones in the development of hardware IFT.

primarily target different HDLs, e.g., Verilog and HLS C, for better scalability and verification performance. In addition, IFT techniques have also been extended to combat analog hardware security exploits.

6 VERIFICATION TECHNIQUE

IFT techniques can operate in a static or dynamic manner. Static hardware IFT techniques [3, 4, 9, 21, 67, 163] check if the design adheres to desired security properties through *simulation*, *formal methods*, *emulation* or *virtual prototyping* during design time. The IFT model will be removed when verification completes. Dynamic hardware IFT [28, 29, 70, 83, 132, 136, 139, 153] is a *runtime* mechanism that augments the original hardware design with tracking logic that monitors information flow behaviors and prevents harmful flows of information. This typically comes at a cost, e.g., additional resource usage and performance overheads.

6.1 Simulation

Simulation is a common method of verifying the behaviors of hardware designs. Functional simulation involves feeding a testbench to a circuit design and observing if the results meet the desired specification. Hardware security simulation plays a similar role except the key output is not functional correctness but rather whether the circuit has violated any security policies. Figure 10 illustrates the difference between functional and IFT based security simulation for identifying information leakage.

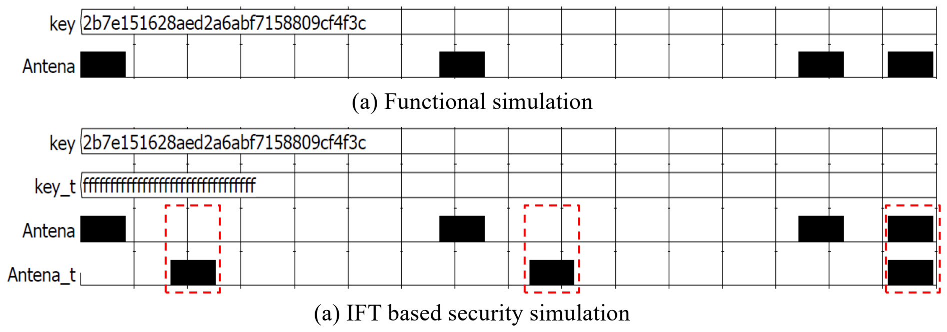

key 2b7e151628aed2a6abf7158809cf4f3c
Antena
(a) Functional simulation
key 2b7e151628aed2a6abf7158809cf4f3c
key_t ffffffffffffffffffffffffffffffff
Antena
Antena_t
(a) IFT based security simulation

Fig. 10. The difference between functional and IFT based security simulation.

In both the functional and security simulation waveforms, there are switching activities in the output signal *Antena* (as spelled in the Trust-HUB AES-T1700 benchmark). However, functional simulation cannot reveal the connection between such switching activities and the key. IFT-based security simulation associates security labels (i.e., *key_t* and *Antena_t*) with the signals, reflecting their security attributes. Here, we label the key as `high`, i.e., *key_t* = 128'hFFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFF. From Fig. 10 (b), the security label *Antena_t* can be logical `1` in the dotted red boxes, indicating that *Antena* can carry `high` information. In other words, *key* leaks to *Antena* during these time slots. The security label *Antena_t* accurately indicates when and where leakage happens even if there might be no switching activity (i.e., leaking a logical `0`) in the *Antena* signal.

Some hardware IFT techniques describe their information flow model using standard HDL (e.g., GLIFT [146] and RTLIFT [3]). In this case, it is possible to perform information flow security simulation using standard or open source EDA simulation tools such as *Mentor Graphics ModelSim or QuestaSim*, *Synopsys VCS*, and *Icarus Verilog*.

Tortuga Logic and *Cadence* have released a commercial hardware security simulation tool, namely the *Radix-S* [98] hardware root of trust security verification framework, which automates the information flow model annotation process for IFT based hardware security simulation using standard HDL and EDA simulation environment.

6.2 Formal Verification

Verification tools can be used for formal (or semi-formal) verification of security properties on information flow models through equivalence checking, SAT solving, theorem proving, or type checking. The benefits of formal approaches are that the hardware is guaranteed to be secure with respect to the properties proven. However, a major downside is that formal methods inherently do not scale due to the exponential size of design state space. Certain properties may not be proved after running for days consuming hundreds of giga bytes of memory, e.g., a hardware Trojan triggered under a specific input pattern or sequence. In addition, the verification outcomes heavily rely on the quality and completeness of the security properties specified.

6.2.1 Equivalence Checking: Equivalence checking is the most frequently used formal verification technique for verifying functional correctness. To check if tainted information propagates from a source signal to a destination point, we can set the taint source to logical `0` and `1` respectively and create two instances of the circuit, i.e., miter circuits. We then check for equivalence at the destination point to see if the taint source has an effect on the destination. This provides an approach to secure path and X-propagation verification [15–17, 44, 46, 94]. These verification tools can be used to prove security properties such as `confidentiality` and `integrity` and identify insecure design paths that could lead to information flow security property violations.

6.2.2 SAT Solving: SAT solvers such as *minSAT* and *zchaff* are effective tools for proving functional properties. Several open source EDA tools such as *ABC* and *Yosys* integrate these SAT solvers and HDL to CNF (conjunctive normal form) formula converters in order to prove design properties on digital circuits. Therefore, they can parse information flow models written in HDL, e.g., IFT models generated by GLIFT [146] and RTLIFT [3], and prove information flow security properties as illustrated in Section 4. Such security properties can be easily converted to proof constraints and incorporated by the solver. The following shows an example of security property mapping for *Yosys* where we constrain the 32-bit *key* to `high` while *clk* to `low` and prove that the *ready* output should always be `low`.

A major difference from simulation is that a counter example will be given when the solver captures a security property violation, indicating exactly when such violation could possibly occur. This provides insightful hints for hardware designers to pinpoint security vulnerabilities.

```
set key := high
assert ready == low
sat -prove ready_t 1'b0 -set key_t 32'hFFFFFFFF -set clk_t 1'b0
```

While SAT tools can be powerful for verifying simple security properties on combinational circuits, they have some inherent limitations as well. They can only verify qualitative security properties on CNF formula. SMT solvers such as *Yices*, on the other hand, does not have this limitation and can prove quantitative security properties on QBF (Quantified Boolean Formula) [7, 52]. In addition, SAT solving on sequential digital circuits can be a challenging task. SAT tools cannot automatically trace beyond register boundaries and thus usually restrict the proof to a minimal combinational region. Unrolling the design by a number of time frames is a frequently used technique for proving properties on sequential designs. However, this technique can lead to huge memory and verification performance overheads.

Commercial formal verification tools can be more effective in verifying security properties and searching for security violations. *Mentor Graphics Questa Formal* [44] takes a semi-formal approach to formal verification. It takes as inputs the initialization vectors and proof-time constraints, simulates the design to a specified proof radius in number of clock cycles and also performs state space search to identify all reachable design states. As shown in Tables 1 to 3, IFT security properties can be specified using property specification language (e.g., SVA and PSL) assertions, which are familiar to hardware designers.

With the security constraints specified in initialization and directive files, the SVA property asserts that at the rising edge of *ready*, the *ready* signal should always be `low`. Existing work has demonstrated that the *Mentor Graphics Questa Formal* [44] tool can be used to prove information flow security properties and identify hard-to-detect security vulnerabilities such as timing channels, insecure debug ports and hardware Trojans [53, 124].

6.2.3 Theorem Proving: A number of hardware IFT methods verify security properties through theorem proving. Information flow security properties are specified as security theorems while security constraints are specified as pre-conditions. Formal tools such as *Coq* can be used to verify these properties [10, 11, 66–68, 123, 124]. There are several HDL annotation tools that convert RTL hardware designs or gate level netlists to formal Coq circuits augmented with security labels [11]. While the formal IFT model construction has largely been automated, the proof process still requires a significant amount of human interaction. New security theorems need to be derived as tainted information reaches new signals, and the designer needs to decide the signals to be checked in the next time frame.

6.2.4 Type Checking: A *Type system* is a set of rules that assigns a property called *type* to each computed value, examines the flow of these values, and attempts to ensure or prove that no type error can occur. A type error occurs when a value of a certain type is used in an operation that is disallowed for its type, e.g., when an `untrusted` variable is used as the jump target in a secure execution context. Type system can be employed for proving security properties by associating variables with security-related types, such as `Confidential` and `Unclassified`. Access or information flow control can be enforced through either static or runtime type checking.

The SecVerilog project [122] releases an open source tool for verifying timing information flow security in Verilog designs. The tool extends Verilog with annotations that support reasoning about information flows including timing flows through compile-time type checking. SecVerilog associates variables with security types (e.g., `{H}` and `{L}`) and labels to specify the information flow policy to be enforced. Verification is done by checking if information flows cause violation of type rules.

6.3 Emulation

With the evolution of IC verification platforms, leading EDA companies have also developed tools to verify information flow security properties of hardware designs on FPGA emulation servers. Hardware emulation can typically achieve 10X+ better verification performance than software simulation [95]. Another advantage of hardware emulation is hardware-software security co-verification, which is not yet possible for formal verification tools. Emulation is a promising approach that leverages the information flow measurement capability of the underlying hardware to perform security co-verification and detect software attacks that exploit hardware security vulnerabilities.

The *Radix-M* [97] hardware security platform from *Tortuga Logic* and *Cadence* is one such emulation tool that performs firmware security validation on full system-on-chip (SoC) designs. The tool deploys hardware IFT models on an FPGA emulation server to run security verifications. Such verification technology can also be used to verify the security of firmware and even software, providing a promising solution to system level security.

6.4 Virtual Prototyping

Virtual prototyping involves creating abstract models/libraries for the different hardware components, which allows verifying the system software before the hardware design is complete. Hassan *et al.* propose an approach for SoC security validation through Virtual Prototypes (VP) modeled using SystemC TLM 2.0 [47, 119]. They employ static information flow analysis to verify the *no flow* security property and detect potential security breaches such as data leakage and untrusted access. High-level abstractive information flow models will lead to better verification performance and scalability. However, they can only employ conservative label propagation policy to profile the possible information flow violations due to the lack of design implementation details at this stage.

6.5 Runtime

Dynamic IFT techniques deploy dedicated hardware to monitor and control information flows during runtime. These dynamic IFT techniques usually target the RTL level or higher, as it becomes costly to perform dynamic IFT at lower levels of abstractions. Additional label memory and IFT logic needs to be integrated into the circuit. At the architectural level, this typically involves a tag propagation unit or co-processor that sits along with the main processor to determine the taint labels of instruction outputs. The labels will then get involved in successive operations or be written back to update the label memory. These techniques usually need to load all the parameters of instructions to the tag propagation unit or co-processor in order to calculate the output taint label. Such parameters can be retrieved either from the execution path of the processor or core debug interface.

Dynamic IFT techniques perform information flow security policy checking on the execution trace, which is typically a small subset of the entire state space. This prevents the state space explosion problem of formal methods and allows the methods to scale to large circuits. However, the additional taint propagation and checking logic typically increases resource usage and power consumption. Even worse, such logic and its synchronization with the original execution context will inevitably cause considerable performance overheads.

As a comparison, formal verification based IFT techniques can completely verify IPs and small subsystems but does not scale to complex system level designs due to the state space explosion problem. Simulation and emulation based IFT approaches have better scalability. However, they cannot cover the entire design state space. Virtual Prototyping allows a quick but conservative profiling of possible information flow security violations. Runtime IFT mechanisms can monitor realistic information flow behaviors but incur large area and performance penalties.

Table 4 summarizes the different hardware IFT techniques in terms of their verification mechanisms. We note that some techniques can be used with different verification techniques, e.g., GLIFT can be used for simulation, emulation, formal verification, and during runtime. Other techniques target a specific area, e.g., VeriCoq [11, 67] and SecVerilog [122, 163] solely use formal verification techniques.

Table 4. Summary of hardware IFT techniques in terms of verification mechanisms used.

Simulation	Caisson[86], Sapper[84], GLIFT[146], GLIFT-SoC[72], GLIFT Timing[113–115], Multi-level GLIFT[54, 55] Multi-value GLIFT[140], Multi-flow IFT[142], Radix-S[98], RTLIFT[3], Clepsydra[2]
Formal Verification	ASSURE[64], GARUDA[131], TaintHLS[120], ChiselFlow[65], Zagieboylo[160], PCH-Coq[10, 66–68], VeriCoq[11] Qin[123, 124], SecVerilog[122, 163], SecVerilogLC[38], Ferraiuolo[39], SecChisel[30], RTLIFT[3], Clepsydra[2], VeriSketch[4] HLIFT[154], Hu[52], Becker[7], GLIFT Timing[113–115], Multi-level GLIFT[54, 55], Multi-value GLIFT[140] Multi-flow IFT[142], Le[81], VeriCoq-analog[9], Guo[45], Nicholas[110], Siddiqui[135], SCRIPT[107]
Emulation	GLIFT[146], RTLIFT[3], Clepsydra[2], Radix-M[97]
Virtual Prototyping	IFT-VP[47], IFT-VP+[119]
Runtime	CDI-IFT[82, 83], PTM-IFT [152], ARMHEx[153], BFBlare[151], PAU[48], IFTU[132], IFT-Accelerator [118, 121] GLIFT-Embedded[103], GLIFT-IoT[21], TaintHLS[120], Saal[127], Bondi[13], DIFT[139], Kannan[70], RIFLE[148], Raksha[25] FlexiTaint[149], Deng[28, 29], HAFIX[26], Shioya[133], Joannou[69], PIFT[92, 93], LATCH[147], DataSafe[20] GLIFT-Lease[87, 143–145], METACE[91], Nagarajan[106], ShadowReplica[63], SIFT[116], Nightingale[111], Chen[19] Ruwase[126], SAFE[5], Nicholas[110], Siddiqui[135], PUMP[35], HDFI[136], HyperFlow[40], Liu[90], Palmiero[117] Shirley[134], Zagieboylo[160], STT[158], Caisson[86], Sapper[84], GLIFT[146], GLIFT-TMR[59, 71]

7 FUTURE RESEARCH DIRECTIONS AND CHALLENGES

7.1 Automated Security Property Generation

Security properties are crucial for hardware IFT verification. High-quality security properties allow verification tools to quickly pinpoint potential security flaws. Unfortunately, specifying security properties that cover a wide range of vulnerabilities is all too often a challenging and daunting process. Crafting hardware security properties requires expertise in formal verification, expertise in security, and knowledge of the design under verification. The job is difficult and time consuming, and critical properties are easily missed. For example, participants of Hack@DAC'18 were able to write properties covering only 17 of the 31 bugs, which "indicates that the main challenge with using formal verification tools is identifying and expressing security properties that the tools are capable of capturing and checking" [34]. Automated generation of high-quality security properties is an important problem, especially for developing property driven hardware security solutions [49, 50]. Two interesting directions in this regard are: 1) developing parameterized security properties and 2) higher level property languages that build upon the existing hardware IFT models, e.g., an access control policy language. We discuss both in the following.

7.2 Parameterized Security Properties

Verification IP (VIP) plays an important role in the hardware design process by providing a set of functional inputs (testbenches, assertions, etc.) for functional verification of a hardware design [125, 159, 165]. The general goal is to determine whether your IP core correctly interfaces with memory protocols (DDR, QSPI, SD), PCIe, USB, ethernet and other communication protocols, and AXI, WISHBONE, and other on-chip interfaces. A similar approach could be done to develop *security VIP* that focuses on security verification rather than functional verification. One potential direction is to develop security property templates based upon information flow metrics [51] and leverage functional property templates from existing assertion based verification approaches [42]. Ideally, the designer would minimally change property parameters based upon their desired security constraints and their hardware design, and the tool would automatically provide a series of relevant properties. Transys provides evidence that this approach is possible [164], but we feel there is much room to generalize and enhance this process.

7.3 Hardware Access Control Policy Language

Modern hardware is comprised of tens to hundreds of different IP cores – processors, custom accelerators, memories, peripherals, timers, and other system resources – that are interconnected through a network of on-chip busses and routers. These IP cores, busses, and memories each have their own security policies which govern how they can behave and how other IP cores can access their data, control registers, and status signals. Since access control is a prevalent and important aspect of hardware security, a property specification language specifically targeted towards access control could allow security experts to specify a wide range of on-chip access control properties that could then be compiled to properties that can be used by existing hardware IFT tools. Additionally, the properties should be able to specify temporal rules, e.g., a memory location cannot be read by certain processes after sensitive information has been written there. Also, they should allow for easy specification of common policies like compartmentalization, access control lists, secure handoff, Chinese wall, and redaction. The properties could utilize lattices to denote allowable accesses. Huffmire et al. defined a language for policies related to off-chip memory controllers [62]. This could be used as a baseline for a more general access control language by modifying it for on-chip resources and provide a clear mapping to hardware security verification properties models.

7.4 Novel Information Flow Models

Hardware IFT opens the door for unique flow models. For example, timing flows become apparent at the RTL [114] and Clepsydra developed a model that updates labels based only on whether there is a timing channel. Note that GLIFT does not do this as it encodes functional and timing models simultaneously (i.e., it can only say a flow occurs, and cannot differentiate if that flow is due to a timing channel or a functional channel). Thus, Clepsydra provided a more expressive IFT model for tracking timing flows. We see opportunities to use IFT for other side channels like power side channel [107], faults, and electromagnetic radiation. We believe this would require more quantitative flow models as opposed to the vast majority of existing models, which take a qualitative approach (there is a flow violation or not). For these side channels, it is often important to know how much information is flowing, in addition to whether there was a flow or not. There is some initial work to develop tools for measuring quantitative information flow properties such as leakage and randomness [49, 142]. We believe that there is more to explore in these directions. Additionally, creating IFT models that can answer questions related to dependability and resilience properties such as availability, reliability, safety and fault tolerance is also an interesting research vector.

7.5 Automated Debugging

Hardware IFT verification often uncovers a security violation. The verification tools may provide a witness, i.e., an example trace describing an execution where the property is violated, but nothing more than that. The onus is on the hardware designer to modify the design and eliminate the flaw. This is a challenging and mostly manual process. Any debugging tools that aid this process provide immense value. Error localization is a fundamental part of this process which identifies the parts of the hardware that need to be modified in order to fix the flaw. If this can be done accurately, one could imagine additional tools to help the verification engineer modify the design to fix the flaws. While fully automated debugging is a huge challenge, there seems to be ample opportunities and value for partially automating the hardware security debugging process.

7.6 Firmware IFT

Firmware is an attractive attack surface for hackers since critical system credentials (e.g., encryption keys and low level access credentials) are stored in or managed by firmware. In addition, firmware provides direct access to the hardware and can be exploited to root the system, uncover confidential system security parameters, and perform other nefarious activities. Firmware requires tracking of information across hardware and software. This is challenging. Firmware IFT verification tools must formalize the flow models for the computational constructs at this level of abstraction. This requires research in combined hardware/software flow models. Subramanyan et al. [138] provide some great initial progress in this space. But there is much more to consider including scalability, mixed hardware/software IFT modeling, and property modeling.

7.7 Analog/Mixed-signal Hardware IFT

Another potential research direction is to move to even lower levels of abstraction and develop IFT techniques for analog or mix-signal hardware designs. Recent work has made a first attempt to develop IFT techniques that allow co-verification of digital and analog hardware to help uncover cross-domain security vulnerabilities [9]. A fundamental problem of performing IFT in analog hardware is label encoding. This is fundamentally different from the digital world since analog signals are continuous signals. Binary labels are no longer eligible for encoding and distinguishing the complex label information. Consequently, we need to define policies for multi-level and/or multi-bit taint label propagation.

7.8 Relationship between IFT and Traditional Function Hardware Verification and Test

Security testing and verification closely relate to several classic problems in switching circuit theory, including controllability, observability, X-propagation, and fault propagation [89]. Taint propagation, X-propagation and conservative fault-propagation can be viewed as the same problem [12, 61]. As an example, *Mentor Graphics* and *Cadence* have both integrated security path and X-propagation Apps into their formal verification tool chain. An interesting research direction is leveraging hardware IFT models to understand these classic testing problems and attempt to formalize them under a unified framework. A recent work has shown how hardware IFT provides a unified model for verifying both taint- and X-propagation properties [61]. On the other hand, existing controllability, observability, X-propagation and fault propagation analysis tools also provide an approach to performing IFT. A benefit of these tools is that they all work on the original circuit model, which has significantly lower complexity than IFT models.

8 CONCLUSION

This article performs a comprehensive review of existing work in the hardware IFT spectrum. Our aim is to draw a clear picture of the development of this technique, bring forward the potential research directions, and discuss the challenges to be resolved. Our intention is to introduce the hardware IFT technique to a wider range of audiences in the hardware security, testing and verification communities and show how this technique can be employed to verify whether a hardware design adheres to both security and reliability properties.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers as well as Jennifer Switzer, graduate researcher at the University of California San Diego and Dr. Lingjuan Wu, associate professor at the Huazhong Agricultural University for their constructive comments and valuable feedback that helped improving this work.

This work was supported in part by the National Natural Science Foundation of China under Grant 62074131 and by the NSF under grant 1718586 and the Semiconductor Research Corporation (SRC).

REFERENCES

[1] Shan Ao and Guo Shuangzhou. 2011. An enhancement technology about system security based on dynamic information flow tracking. In *Int. Conf. on Artif. Intell., Manag. Sci. and Elec. Commerce (AIMSEC)* (Zhengzhou, China). 6108–6111.

[2] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. 2017. Clepsydra: Modeling timing flows in hardware designs. In *Int. Conf. Comput.-Aided Des. (ICCAD)* (Irvine, CA, USA). 147–154.

[3] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Des. Autom. Test Europe Conf. Exhib. (DATE)* (Lausanne, Switzerland). 1691–1696.

[4] Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. 2019. VeriSketch: Synthesizing Secure Hardware Designs with Timing-Sensitive Information Flow Properties. In *ACM Conf. on Comp. and Comm. Sec. (CCS)* (London, United Kingdom). ACM, 1623–1638.

[5] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2014. A Verified Information-Flow Architecture. In *SIGPLAN-SIGACT Symp. on Principles of Prog. Lang. (POPL)* (San Diego, CA, USA). ACM, 165âĂŞ178.

[6] Jean Bacon, David Eyers, Thomas FJ-M Pasquier, Jatinder Singh, Ioannis Papagiannis, and Peter Pietzuch. 2014. Information flow control for secure cloud computing. *IEEE Trans. Netw. Serv. Manag.* 11, 1 (2014), 76–89.

[7] Andrew Becker, Wei Hu, Yu Tai, Philip Brisk, Ryan Kastner, and Paolo Ienne. 2017. Arbitrary precision and complexity tradeoffs for gate-level information flow tracking. In *Des. Autom. Conf. (DAC)* (Austin, TX, USA). 1–6.

[8] Lionel Bening and Harry Foster. 2001. *Principles of Verifiable RTL Design* (2nd ed.). Kluwer Academic Publishers, New York, USA.

[9] Mohammad-Mahdi Bidmeshki, Angelos Antonopoulos, and Yiorgos Makris. 2017. Information flow tracking in analog/mixed-signal designs through proof-carrying hardware IP. In *Des. Autom. Test Europe Conf. Exhib. (DATE)* (Lausanne, Switzerland). 1703–1708.

[10] Mohammad-Mahdi Bidmeshki, Xiaolong Guo, Raj Gautam Dutta, Yier Jin, and Yiorgos Makris. 2017. Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IPâĂŤPart II: Framework Automation. *IEEE Trans. Inf. Forensics Security* 12, 10 (Oct 2017), 2430–2443.

[11] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. 2015. Toward automatic proof generation for information flow policies in third-party hardware IP. In *IEEE Int. Symp. on Hardware Oriented Security and Trust (HOST)* (Washington, DC, USA). 163–168.

[12] Jeremy Blackstone, Wei Hu, Alric Althoff, Armaiti Ardeshiricham, Lu Zhang, and Ryan Kastner. 2020. A Unified Model for Gate Level Propagation Analysis. *ArXiv e-prints* (Dec. 2020). arXiv:2012.02791

[13] James O. Bondi and Martha A. Branstad. 1989. Architectural support of fine-grained secure computing. In *Comp. Sec. App. Conf.* 121–130.

[14] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symp. (USENIX Security)* (Baltimore, MD, USA). USENIX Association, 991–1008.

[15] G. Cabodi, P. Camurati, S. F. Finocchiaro, C. Loiacono, F. Savarese, and D. Vendraminetto. 2016. Secure Path Verification. In *IEEE Int. Verification and Security Workshop (IVSW)* (St. Feliu de Guixols, Spain). 1–6.

[16] G. Cabodi, P. Camurati, S. F. Finocchiaro, F. Savarese, and D. Vendraminetto. 2017. Embedded Systems Secure Path Verification at the Hardware/Software Interface. *IEEE Design & Test* 34, 5 (Oct 2017), 38–46.

[17] Cadence. 2016. JasperGold Security Path Verification App. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html.

[18] Andrew Canis, Jongsok Choi, Blair Fort, Bain Syrowik, Ruo Long Lian, Yu Ting Chen, Hsuan Hsiao, Jeffrey Goeders, Stephen Brown, and Jason Anderson. 2016. *LegUp High-Level Synthesis.* Springer, Cham, Switzerland, 175–190.

[19] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. 2008. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Int. Symp. on Comp. Archit. (ISCA)* (Beijing, China). IEEE, 377âĂŞ388.

[20] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. 2012. A Software-hardware Architecture for Self-protecting Data. In *ACM Conf. on Comp. and Comm. Sec. (CCS)* (Raleigh, NC, USA). ACM, 14–27.

[21] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Software-based Gate-level Information Flow Security for IoT Systems. In *Int. Symp. on Microarchitecture (MICRO)* (Boston, MA, USA). ACM, 328–340.

[22] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *J. Comp. Security* 18, 6 (2010), 1157–1210.

[23] Ellis Cohen. 1977. Information transmission in computational systems. *ACM SIGOPS Oper. Syst. Rev.* 11, 5 (1977), 133–139.

[24] Ellis S Cohen. 1978. Information transmission in sequential programs. *Foundations of Secure Computation* (1978), 297–335.

[25] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. *ACM SIGARCH Comput. Archit. News* 35, 2 (June 2007), 482âĂŞ493.

[26] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-Assisted Flow Integrity eXtension. In *Des. Autom. Conf. (DAC)* (San Francisco, CA, USA). 1–6.

[27] Onur Demir, Wenjie Xiong, Faisal Zaghloul, and Jakub Szefer. 2016. Survey of Approaches for Security Verification of Hardware/Software Systems. Cryptology ePrint Archive, Report 2016/846. https://eprint.iacr.org/2016/846.

[28] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. 2010. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Int. Symp. on Microarchitecture (MICRO)* (Atlanta, GA, USA). 137–148.

[29] Daniel Y. Deng and G. Edward Suh. 2012. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *IEEE/IFIP Int. Conf. on Depend. Syst. and Netw. (DSN)* (Boston, MA, USA). 1–12.

[30] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Sercan Sari, Y. Serhan Gener, Corine Lu, Onur Demir, and Jakub Szefer. 2019. SecChisel Framework for Security Verification of Secure Processor Architectures. In *Int. Workshop on Hardw. and Archit. Supp. for Sec. and Priv. (HASP)* (Phoenix, AZ, USA). ACM, Article 7, 8 pages.

[31] Dorothy Elizabeth Robling Denning. 1975. *Secure Information Flow in Computer Systems.* Ph.D. Dissertation. West Lafayette, IN, USA. AAI7600514.

[32] Dorothy Elizabeth Robling Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (1976), 236–243.

[33] Jack B Dennis and Earl C Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155.

[34] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2019. Hardfails: Insights into software-exploitable hardware bugs. In *USENIX Security Symp. (USENIX Security).* 213–230.

[35] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and Andre DeHon. 2015. Architectural Support for Software-Defined Metadata Processing. *ACM SIGPLAN Not.* 50, 4 (March 2015), 487âĂŞ502.

[36] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. 2005. Labels and event processes in the Asbestos operating system. *ACM SIGOPS Oper. Syst. Rev.* 39, 5 (2005), 17–30.

[37] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic spyware analysis. In *USENIX Annual Technical Conference.* USENIX Association.

[38] Andrew Ferraiuolo, Weizhe Hua, Andrew C. Myers, and G. Edward Suh. 2017. Secure information flow verification with mutable dependent types. In *Des. Autom. Conf. (DAC)* (Austin, TX, USA). 1–6.

[39] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2017. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. *ACM SIGARCH Comput. Archit. News* 45, 1 (April 2017), 555âĂŞ568.

[40] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. 2018. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *ACM Conf. on Comp. and Comm. Sec. (CCS)* (Toronto, Canada). ACM, 1583âĂŞ1600.

[41] Harry Foster. 2009. *Applied assertion-based verification: An industry perspective.* Now Publishers Inc, Boston, MA, USA.

[42] Harry Foster, Adam Krolnik, and David Lacey. 2005. *Assertion-Based Design.* Springer, Boston, MA, USA. 61–102 pages.

[43] Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *IEEE Symp. on Security & Privacy.* 11–20.

[44] Mentor Graphics. 2016. Questa secure check - exhaustive verification of secure paths to critical hardware storage. https://www.mentor.com/products/fv/questa-secure-check.

[45] Xiaolong Guo, Huifeng Zhu, Yier Jin, and Xuan Zhang. 2019. When Capacitors Attack: Formal Method Driven Design and Detection of Charge-Domain Trojans. In *Des. Autom. Test Europe Conf. Exhib. (DATE)* (Florence, Italy). 1727–1732.

[46] Ziyad Hanna. 2013. Jasper case study on formally verifying secure on-chip datapaths. http://www.deepchip.com/items/0524-03.html.

[47] Muhammad Hassan, Vladimir Herdt, Hoang M. Le, Daniel Große, and Rolf Drechsler. 2017. Early SoC Security Validation by VP-Based Static Information Flow Analysis. In *Int. Conf. Comput.-Aided Des. (ICCAD)* (Irvine, CA, USA). IEEE, 400âĂŞ407.

[48] Ingoo Heo, Minsu Kim, Yongje Lee, Changho Choi, Jinyong Lee, Brent Byunghoon Kang, and Yunheung Paek. 2015. Implementing an Application-Specific Instruction-Set Processor for System-Level Dynamic Program Analysis Engines. *ACM Trans. Des. Autom. Electron. Syst.* 20, 4, Article 53 (Sept. 2015), 32 pages.

[49] Wei Hu, Alric Althoff, Armaiti Ardeshiricham, and Ryan Kastner. 2016. Towards Property Driven Hardware Security. In *Int. Workshop on Microprocessor and SOC Test and Verification (MTV)* (Austin, TX, USA). 51–56.

[50] Wei Hu, Armaiti Ardeshiricham, Mustafa S Gobulukoglu, Xinmu Wang, and Ryan Kastner. 2018. Property specific information flow analysis for hardware security verification. In *Int. Conf. Comput.-Aided Des. (ICCAD)* (San Diego, CA, USA). IEEE, 1–8.

[51] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2017. Identifying and measuring security critical path for uncovering circuit vulnerabilities. In *Int. Workshop on Microprocessor and SOC Test and Verification (MTV)* (Austin, TX, USA). IEEE, 62–67.

[52] Wei Hu, Andrew Becker, Armita Ardeshiricham, Yu Tai, Paolo Ienne, Dejun Mu, and Ryan Kastner. 2016. Imprecise security: Quality and complexity tradeoffs for hardware information flow tracking. In *Int. Conf. Comput.-Aided Des. (ICCAD)* (Austin, TX, USA). 1–8.

[53] Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. 2016. Detecting Hardware Trojans with Gate-Level Information-Flow Tracking. *Computer* 49, 8 (Aug 2016), 44–52.

[54] Wei Hu, Dejun Mu, Jason Oberg, Baolei Mao, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. 2014. Gate-Level Information Flow Tracking for Security Lattices. *ACM Trans. Des. Autom. Electron. Syst.* 20, 1, Article 2 (Nov. 2014), 25 pages.

[55] Wei Hu, Jason Oberg, Jeremy Barrientos, Dejun Mu, and Ryan Kastner. 2013. Expanding Gate Level Information Flow Tracking for Multilevel Security. *IEEE Embedd. Syst. Lett.* 5, 2 (June 2013), 25–28.

[56] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, , and Ryan Kastner. 2011. An Improved Encoding Technique for Gate Level Information Flow Tracking. In *Int. Workshop on Logic & Synthesis (IWLS)* (San Diego, CA, USA).

[57] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. 2011. Theoretical Fundamentals of Gate Level Information Flow Tracking. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 30, 8 (Aug 2011), 1128–1140.

[58] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. 2012. On the Complexity of Generating Gate Level Information Flow Tracking Logic. *IEEE Trans. Inf. Forensics Security* 7, 3 (June 2012), 1067–1080.

[59] Wei Hu, Jason Oberg, Dejun Mu, and Ryan Kastner. 2012. Simultaneous Information Flow Security and Circuit Redundancy in Boolean Gates. In *Int. Conf. Comput.-Aided Des. (ICCAD)* (San Jose, CA, USA). ACM, 585–590.

[60] Wei Hu, Xinmu Wang, and Dejun Mu. 2018. Security Path Verification Through Joint Information Flow Analysis. In *IEEE Asia Pacific Conf. on Circ. and Syst. (APCCAS)* (Chengdu, China). 415–418.

[61] W. Hu, L. Wu, Y. Tai, J. Tan, and J. Zhang. 2020. A Unified Formal Model for Proving Security and Reliability Properties. In *IEEE Asian Test Symp. (ATS)* (Penang, Malaysia). 1–6.

[62] Ted Huffmire, Shreyas Prasad, Tim Sherwood, and Ryan Kastner. 2006. Policy-Driven Memory Protection for Reconfigurable Hardware. In *European Conference on Research in Computer Security* (Hamburg, Germany) *(ESORICS'06)*. Springer-Verlag, Berlin, Heidelberg, 461âĂŞ478.

[63] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. 2013. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking. In *ACM Conf. on Comp. and Comm. Sec. (CCS)* (Berlin, Germany). ACM, 235âĂŞ246.

[64] Zhenghong Jiang, Steve Dai, G. Edward Suh, and Zhiru Zhang. 2018. High-Level Synthesis with Timing-Sensitive Information Flow Enforcement. In *Int. Conf. Comput.-Aided Des. (ICCAD)* (San Diego, CA, USA). ACM, Article 88, 8 pages.

[65] Zhenghong Jiang, Hanchen Jin, G. Edward Suh, and Zhiru Zhang. 2019. Designing Secure Cryptographic Accelerators with Information Flow Enforcement: A Case Study on AES. In *Des. Autom. Conf. (DAC)* (Las Vegas, NV, USA). ACM, Article 59, 6 pages.

[66] Yier Jin, Xiaolong Guo, Raj Gautam Dutta, Mohammad-Mahdi Bidmeshki, and Yiorgos Makris. 2017. Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IPâĂŤPart I: Framework Fundamentals. *IEEE Trans. Inf. Forensics Security* 12, 10 (Oct 2017), 2416–2429.

[67] Yier Jin and Yiorgos Makris. 2012. Proof carrying-based information flow tracking for data secrecy protection and hardware trust. In *IEEE VLSI Test Symp. (VTS)* (Maui, HI, USA). 252–257.

[68] Yier Jin, Bo Yang, and Yiorgos Makris. 2013. Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing. In *IEEE Int. Symp. on Hardware Oriented Security and Trust (HOST)* (Austin, TX, USA). 99–106.

[69] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *IEEE Int. Conf. on Comp. Des. (ICCD)* (Boston, MA, USA). 641–648.

[70] Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2009. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *IEEE/IFIP Int. Conf. on Depend. Syst. and Netw. (DSN)* (Lisbon, Portugal). 105–114.

[71] Ryan Kastner. 2013. Circuit Primitives for Monitoring Information Flow and Enabling Redundancy. In *Int. Conf. on Hardware and Software: Verification and Testing (HVC)* (Haifa, Israel). Springer-Verlag, Berlin, Heidelberg, 6–6.

[72] Ryan Kastner, Jason Oberg, Wei Hu, and Ali Irturk. 2011. Enforcing Information Flow Guarantees in Reconfigurable Systems with Mix-trusted IP. In *Int. Conf. on Engineering of Reconfig. Syst. and Algorithm. (ERSA)* (Las Vegas, NV, USA).

[73] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. *SIGARCH Comput. Archit. News* 42, 3 (2014), 361âĂŞ372.

[74] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *the ACM SIGOPS Symp. on Oper. Syst. Princip. (SOSP)* (Big Sky, MT, USA). ACM, 207âĂŞ220.

[75] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[76] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99*, Michael Wiener (Ed.). Springer, Berlin, Heidelberg, 388–397.

[77] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, Neal Koblitz (Ed.). Springer, Berlin, Heidelberg, 104–113.

[78] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. *ACM SIGOPS Oper. Syst. Rev.* 41, 6 (2007), 321–334.

[79] Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (Oct. 1973), 613–615.

[80] Butler W Lampson. 1974. Protection. *ACM SIGOPS Oper. Syst. Rev.* 8, 1 (1974), 18–24.

[81] Thao Le, Jia Di, Mark Tehranipoor, and Lei Wang. 2016. Tracking data flow at gate-level through structural checking. In *Int. Great Lakes Symp. on VLSI (GLSVLSI)* (Boston, MA, USA). 185–189.

[82] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. 2015. Efficient dynamic information flow tracking on a processor with core debug interface. In *Des. Autom. Conf. (DAC)* (San Francisco, CA, USA). 1–6.

[83] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. 2016. Efficient Security Monitoring with the Core Debug Interface in an Embedded Processor. *ACM Trans. Des. Autom. Electron. Syst.* 22, 1, Article 8 (May 2016), 29 pages.

[84] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2013. Position Paper: Sapper – a Language for Provable Hardware Policy Enforcement. In *SIGPLAN Workshop on Prog. Lang. and Analysis for Sec. (PLAS)* (Seattle, WA, USA). ACM, 39–44.

[85] Xun Li, Mohit Tiwari, Ben Hardekopf, Timothy Sherwood, and Frederic T. Chong. 2010. Secure Information Flow Analysis for Hardware Design: Using the Right Abstraction for the Job. In *SIGPLAN Workshop on Prog. Lang. and Analysis for Sec.* (Toronto, Canada). ACM, Article 8, 7 pages.

[86] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: A Hardware Description Language for Secure Information Flow. In *SIGPLAN Conf. on Prog. Lang. Des. and Impl. (PLDI)* (San Jose, CA, USA). ACM, 109–120.

[87] Xun Li, Mohit Tiwari, Timothy Sherwood, and Frederic T. Chong. 2010. Function flattening for lease-based, information-leak-free systems. In *IEEE Int. Conf. on App.-specific Syst., Archit. and Processors (ASAP)* (Rennes, France). 349–352.

[88] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[89] Peter Lisherness and Kwang-Ting Cheng. 2009. An instrumented observability coverage method for system validation. In *IEEE Int. High Level Des. Validation and Test Workshop* (San Francisco, CA, USA). 88–93.

[90] Jed Liu, Joe Corbett-Davies, Andrew Ferraiuolo, Alexander Ivanov, Mulong Luo, G. Edward Suh, Andrew C. Myers, and Mark Campbell. 2018. Secure Autonomous Cyber-Physical Systems Through Verifiable Information Flow Control. In *Cyber-Physical Syst. Sec. and PrivaCy (CPS-SPC)* (Toronto, Canada). ACM, 48âĂŞ59.

[91] Juan Carlos Martínez Santos and Yunsi Fei. 2013. Micro-Architectural Support for Metadata Coherence in Multi-Core Dynamic Information Flow Tracking. In *Int. Workshop on Hardw. and Arch. Supp. for Sec. and Priv. (HASP)* (Tel-Aviv, Israel). ACM, Article 6, 8 pages.

[92] Juan Carlos Martinez Santos, Yunsi Fei, and Zhijie Jerry Shi. 2009. PIFT: Efficient Dynamic Information Flow Tracking Using Secure Page Allocation. In *Workshop on Embedd. Syst. Sec. (WESS)* (Grenoble, France). ACM, Article 6, 8 pages.

[93] Juan Carlos Martinez Santos, Yunsi Fei, and Zhijie Jerry Shi. 2012. Static Secure Page Allocation for Light-Weight Dynamic Information Flow Tracking. In *Int. Conf. on Compil., Archit. and Synth. for Embedded Syst. (CASES)* (Tampere, Finland). ACM, 27âĂŞ36.

[94] Jamil Mazzawi and Ziyad Hanna. 2013. Formal Analysis of Security Data Paths in RTL Design. In *Hardware and Software: Verification and Testing*, Armin Biere, Amir Nahir, and Tanja Vos (Eds.). Springer, Berlin, Heidelberg, 7–7.

[95] Mentor Graphics. 2015. From Simulation to Emulation âĂŞ A Fully Reusable UVM Framework. https://www.mentor.com/products/fv/resources/overview/from-simulation-to-emulation-a-fully-reusable-uvm-framework-0def891c-ab7a-453d-b079-2c99f584650e

[96] MITRE. 2020. Common Weakness Enumeration (CWE). https://cwe.mitre.org/.

[97] Tortuga Logic. 2019. Radix-M hardware security platform for firmware security validation. https://www.tortugalogic.com/radix-m/.

[98] Tortuga Logic. 2019. Radix-S Hardware Root of Trust security verification framework. https://www.tortugalogic.com/radix-s/.

[99] Tortuga Logic. 2020. Measurable Hardware Security with MITRE CWEs. https://tortugalogic.com/tech-info/.

[100] US Department of Defense. 1985. *Department of Defense Trusted Computer System Evaluation Criteria.* Palgrave Macmillan UK, London, 1–129.

[101] John McLean. 1992. Proving noninterference and functional correctness using traces. *J. Comp. Security* 1, 1 (1992), 37–57.

[102] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring multiple execution paths for malware analysis. In *IEEE Symp. on Security and Privacy (SP)* (Berkeley, CA, USA). IEEE, 231–245.

[103] Dejun Mu, Wei Hu, Baolei Mao, and Bo Ma. 2014. A bottom-up approach to verifiable embedded system information flow security. *IET Information Security* 8, 1 (Jan 2014), 12–17.

[104] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *IEEE Symp. on Security and Privacy (SP)* (Berkeley, CA, USA). 415–429.

[105] Andrew C Myers and Barbara Liskov. 1997. A decentralized model for information flow control. *ACM SIGOPS Oper. Syst. Rev.* 31, 5 (1997), 129–142.

[106] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. 2008. Dynamic Information Flow Tracking on Multicores. In *Workshop on Interac. between Compil. and Comp. Archit.* (Salt Lake City, UT, USA).

[107] Adib Nahiyan, Jungmin Park, Miao He, Yousef Iskander, Farimah Farahmandi, Domenic Forte, and Mark Tehranipoor. 2020. SCRIPT: A CAD Framework for Power Side-Channel Vulnerability Assessment Using Information Flow Tracking and Pattern Generation. *ACM Trans. Des. Autom. Electron. Syst.* 25, 3, Article 26 (May 2020), 27 pages.

[108] Adib Nahiyan, Mehdi Sadi, Rahul Vittal, Gustavo Contreras, Domenic Forte, and Mark Tehranipoor. 2017. Hardware trojan detection through information flow security verification. In *IEEE Int. Test Conf. (ITC)* (Fort Worth, TX, USA). 1–10.

[109] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Netw. and Distrib. Syst. Sec. Symp. (NDSS)* (San Diego, CA, USA), Vol. 5. Citeseer, 3–4.

[110] Geraldine Shirley Nicholas, Yutian Gui, and Fareena Saqib. 2020. A Survey and Analysis on SoC Platform Security in ARM, Intel and RISC-V Architecture. In *IEEE Int. Midwest Symp. on Circ. and Syst. (MWSCAS)* (Springfield, MA, USA). 718–721.

[111] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. 2008. Parallelizing Security Checks on Commodity Hardware. In *Int. Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys. (ASPLOS)* (Seattle, WA, USA). ACM, 308âĂŞ318.

[112] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. 2010. Theoretical analysis of gate level information flow tracking. In *Des. Autom. Conf. (DAC)* (San Jose, CA, USA). 244–247.

[113] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. 2011. Information Flow Isolation in I2C and USB. In *Des. Autom. Conf. (DAC)* (San Diego, CA, USA). ACM, 254–259.

[114] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. 2014. Leveraging Gate-Level Properties to Identify Hardware Timing Channels. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 33, 9 (Sept 2014), 1288–1301.

[115] Jason Oberg, Timothy Sherwood, and Ryan Kastner. 2013. Eliminating Timing Information Flows in a Mix-Trusted System-on-Chip. *IEEE Design & Test* 30, 2 (April 2013), 55–62.

[116] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. 2014. SIFT: Low-Complexity Energy-Efficient Information Flow Tracking on SMT Processors. *IEEE Trans. Comput.* 63, 2 (Feb 2014), 484–496.

[117] Christian Palmiero, Giuseppe Di Guglielmo, Luciano Lavagno, and Luca P. Carloni. 2018. Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications. In *IEEE High Perform. Extreme Comput. Conf. (HPEC)*. 1–7.

[118] Luca Piccolboni, Giuseppe Di Guglielmo, and Luca P. Carloni. 2018. PAGURUS: Low-Overhead Dynamic Information Flow Tracking on Loosely Coupled Accelerators. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 37, 11 (Nov 2018), 2685–2696.

[119] Pascal Pieper, Vladimir Herdt, Daniel GroÃ§e, and Rolf Drechsler. 2020. Dynamic Information Flow Tracking for Embedded Binaries using SystemC-based Virtual Prototypes. In *Des. Autom. Conf. (DAC)* (San Francisco, CA, USA). 1–6.

[120] Christian Pilato, Kaijie Wu, Siddharth Garg, Ramesh Karri, and Francesco Regazzoni. 2019. TaintHLS: High-Level Synthesis for Dynamic Information Flow Tracking. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 38, 5 (May 2019), 798–808.

[121] Joël Porquet and Simha Sethumadhavan. 2013. WHISK: An Uncore Architecture for Dynamic Information Flow Tracking in Heterogeneous Embedded SoCs. In *Int. Conf. on Hardware/Software Codesign and Syst. Synth. (CODES+ISSS)* (Quebec, Canada). IEEE, Article 4, 9 pages.

[122] SecVerilog Project. 2015. SecVerilog: Verilog + Information Flow V1.0. http://www.cs.cornell.edu/projects/secverilog/.

[123] Maoyuan Qin, Wei Hu, Dejun Mu, and Yu Tai. 2018. Property Based Formal Security Verification for Hardware Trojan Detection. In *IEEE Int. Verification and Security Workshop (IVSW)* (Costa Brava, Spain). IEEE, 62–67.

[124] Maoyuan Qin, Wei Hu, Xinmu Wang, Dejun Mu, and Baolei Mao. 2019. Theorem proof based gate level information flow tracking for hardware security verification. *Computers & Security* 85 (2019), 225 – 239.

[125] Sharon Rosenberg and Kathleen Meade. 2013. *A practical guide to adopting the universal verification methodology (UVM)*. Cadence.

[126] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. 2008. Parallelizing Dynamic Information Flow Tracking. In *Symp. on Parall. in Algorithm. and Archit. (SPAA)* (Munich, Germany). ACM, 35âĂŞ45.

[127] Harry J. Saal and Israel Gat. 1978. A Hardware Architecture for Controlling Information Flow. In *Int. Symp. on Comp. Archit. (ISCA)*. ACM, 73–77.

[128] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. on Sel. Areas in Comm.* 21, 1 (Jan 2003), 5–19.

[129] Hassan Salmani, Mohammad Tehranipoor, and Ramesh Karri. 2013. On design vulnerability analysis and trust benchmarks development. In *IEEE Int. Conf. on Comp. Des. (ICCD)* (Asheville, NC, USA). 471–474.

[130] Jerome H Saltzer, David P Reed, and David D Clark. 1984. End-to-end arguments in system design. *Technology* 100 (1984), 0661.

[131] Seaghan Sefton, Taiman Siddiqui, Nathaniel St. Amour, Gordon Stewart, and Avinash Karanth Kodi. 2018. GARUDA: Designing Energy-Efficient Hardware Monitors From High-Level Policies for Secure Information Flow. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 37, 11 (Nov 2018), 2509–2518.

[132] Jangseop Shin, Hongce Zhang, Jinyong Lee, Ingoo Heo, Yu-Yuan Chen, Ruby Lee, and Yunheung Paek. 2016. A hardware-based technique for efficient implicit information flow tracking. In *Int. Conf. Comput.-Aided Des. (ICCAD)* (Austin, TX, USA). 1–7.

[133] Ryota Shioya, Daewung Kim, Kazuo Horio, Masahiro Goshima, and Shuichi Sakai. 2009. Low-Overhead Architecture for Security Tag. In *IEEE Pacific Rim Int. Symp. on Depend. Computing* (Shanghai, China). 135–142.

[134] Geraldine Shirley and Fareena Saqib. 2019. Information Flow Tracking in RISC-V. In *IEEE Int. Conf. on Smart Cities: Improving Quality of Life Using ICT IoT and AI (HONET-ICT)* (Charlotte, NC, USA). 199–200.

[135] Ali Shuja Siddiqui, Geraldine Shirley, Shreya Bendre, Girija Bhagwat, Jim Plusquellic, and Fareena Saqib. 2019. Secure Design Flow of FPGA Based RISC-V Implementation. In *IEEE Int. Verification and Security Workshop (IVSW)* (Rhodes Island, Greece). 37–42.

[136] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *IEEE Symp. on Security and Privacy (SP)* (San Jose, CA, USA). 1–17.

[137] Pramod Subramanyan and Divya Arora. 2014. Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design. In *Des. Autom. Test Europe Conf. Exhib. (DATE)* (Dresden, Germany). IEEE, Article 313, 2 pages.

[138] Pramod Subramanyan, Sharad Malik, Hareesh Khattri, Abhranil Maiti, and Jason Fung. 2016. Verifying information flow properties of firmware using symbolic execution. In *Des. Autom. Test Europe Conf. Exhib. (DATE)* (Dresden, Germany). 337–342.

[139] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Int. Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys. (ASPLOS)* (Boston, MA, USA). ACM, 85–96.

[140] Yu Tai, Wei Hu, Lantian Guo, Baolei Mao, and Dejun Mu. 2017. Gate Level Information Flow analysis for multi-valued logic system. In *Int. Conf. on Image, Vision and Computing (ICIVC)* (Chengdu, China). 1102–1106.

[141] Yu Tai, Wei Hu, Dejun Mu, Baolei Mao, and Lu Zhang. 2018. Towards Quantified Data Analysis of Information Flow Tracking for Secure System Design. *IEEE Access* 6 (2018), 1822–1831.

[142] Yu Tai, Wei Hu, Lu Zhang, Dejun Mu, and Ryan Kastner. 2021. A multi-flow information flow tracking approach for proving quantitative hardware security properties. *Tsinghua Sci. and Tech.* 26, 1 (2021), 62–71.

[143] Mohit Tiwari, Xun Li, Hassan M G Wassel, Frederic T Chong, and Timothy Sherwood. 2009. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Int. Symp. on Microarchitecture (MICRO)* (New York, NY, USA). 493–504.

[144] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2010. Gate-Level Information-Flow Tracking for Secure Architectures. *IEEE Micro* 30, 1 (Jan 2010), 92–100.

[145] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2011. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Int. Symp. on Comp. Archit. (ISCA)* (San Jose, CA, USA). 189–199.

[146] Mohit Tiwari, Hassan M. G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In *Int. Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys. (ASPLOS)* (Washington, DC, USA). 109–120.

[147] Daniel Townley, Khaled N. Khasawneh, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Lei Yu. 2019. LATCH: A Locality-Aware Taint CHecker. In *Int. Symp. on Microarchitecture (MICRO)* (Columbus, OH, USA). ACM, 969âĂŞ982.

[148] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. 2004. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Int. Symp. on Microarchitecture (MICRO)* (Portland, OR, USA). 243–254.

[149] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *IEEE Int. Symp. on High Perform. Comp. Archit.* (Salt Lake City, UT, USA). 173–184.

[150] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *J. Comp. Security* 4, 2-3 (1996), 167–187.

[151] Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Arnab Kumar Biswas, Vianney LapÃŤtre, and Guy Gogniat. 2018. A small and adaptive coprocessor for information flow tracking in ARM SoCs. In *Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)* (Cancun, Mexico). 1–8.

[152] Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Vianney Lapotre, and Guy Gogniat. 2016. Towards a hardware-assisted information flow tracking ecosystem for ARM processors. In *Int. Conf. on Field Programm. Logic and App. (FPL)* (Lausanne, Switzerland). 1–2.

[153] Muhammad A. Wahab, Pascal Cotret, Mounir N. Allah, Guillaume Hiet, Vianney LapÃťtre, and Guy Gogniat. 2017. ARMHEx: A hardware extension for DIFT on ARM-based SoCs. In *Int. Conf. on Field Programm. Logic and App. (FPL)* (Ghent, Belgium). 1–7.

[154] Chenguang Wang, Yici Cai, and Qiang Zhou. 2018. HLIFT: A high-level information flow tracking method for detecting hardware Trojans. In *Asia and South Pacific Des. Autom. Conf. (ASP-DAC)* (Jeju, South Korea). 727–732.

[155] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. https://foreshadowattack.eu/foreshadow-NG.pdf

[156] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson, and Fred Pollack. 1974. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM* 17, 6 (1974), 337–345.

[157] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. 2016. A2: Analog Malicious Hardware. In *IEEE Symp. on Security and Privacy (SP)* (San Jose, CA, USA). 18–37.

[158] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Int. Symp. on Microarchitecture (MICRO)* (Columbus, OH, USA). ACM, 954âĂŞ968.

[159] Young-Nam Yun, Jae-Beom Kim, Nam-Do Kim, and Byeong Min. 2011. Beyond UVM for practical SoC verification. In *Int. SoC Des. Conf.* 158–162.

[160] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. 2019. Using Information Flow to Design an ISA that Controls Timing Channels. In *IEEE Comp. Sec. Foundations Symp. (CSF)* (Hoboken, NJ, USA). 272–27215.

[161] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2011. Making information flow explicit in HiStar. *Commun. ACM* 54, 11 (2011), 93–101.

[162] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. 2008. Securing Distributed Systems with Information Flow Control. In *Netw. Syst. Des. & Impl. (NSDI)* (San Francisco, CA, USA), Vol. 8. 293–308.

[163] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Int. Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys. (ASPLOS)* (Istanbul, Turkey). ACM, 503–516.

[164] Rui Zhang and Cynthia Sturton. 2020. Transys: Leveraging Common Security Properties Across Hardware Designs. In *IEEE Symp. on Security and Privacy (SP)* (San Francisco, CA, USA). 1713–1727.

[165] Hu Zhaohui, Arnaud Pierres, Hu Shiqing, Chen Fang, Philippe Royannez, Eng Pek See, and Yean Ling Hoon. 2012. Practical and efficient SOC verification flow by reusing IP testcase and testbench. In *Int. SoC Des. Conf. (ISOCC)* (Jeju Island, South Korea). IEEE, 175–178.