UC San Diego
UC San Diego Previously Published Works

Title
Information Flow Verification

Permalink
https://escholarship.org/uc/item/02c6z3s3

ISBN
9789819793136

Authors
Sturton, Cynthia
Kastner, Ryan

Publication Date
2025

DOI
10.1007/978-981-97-9314-3_42

Copyright Information
This work is made available under the terms of a Creative Commons Attribution-NonCommercial-NoDerivatives License, available at
https://creativecommons.org/licenses/by-nc-nd/4.0/

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Information Flow Verification

Cynthia Sturton
University of North Carolina at Chapel Hill
csturton@cs.unc.edu

Ryan Kastner
University of California San Diego
kastner@ucsd.edu

Abstract

Information flow tracking (IFT) models the movement of data which enables verification of security properties related to integrity and confidentiality. This chapter introduces the basics of hardware information flow analysis and illustrates its use for hardware security verification. The chapter starts by describing information flow models and properties. Then it highlights how information flow analysis is critical to hardware security verification. After that, there is a discussion regarding the differences between trace properties, which are commonly used in hardware functional verification, and information-flow properties. This is followed by a description of different information flow verification techniques. The chapter concludes with two case studies demonstrating security verification using information flow properties.

Keywords Information flow model; hardware security properties; hardware design verification; cache timing side channels; access control properties

1 Introduction

Security verification plays a crucial role in the development of modern hardware. Hardware vulnerabilities are difficult and expensive to fix if they are not caught early. Additionally, many security features are implemented in hardware, e.g., encryption, key management, authentication, and other hardware root of trust operations. Thus, semiconductor manufacturers and system integrators are increasing the amount of time and effort spent on hardware security verification.

Information flow tracking (IFT) is a fundamental technique that models how data propagates throughout a system. IFT verification aims to determine that only authorized flows of information are possible. IFT has been used to verify security of the cloud [1], operating systems [2, 3, 4], programming languages [5], and hardware [6].

Hardware IFT allows verification of a wide range of security properties related to confidentiality, integrity, timing, availability, safety, hardware Trojans, and speculative execution [6]. The key idea behind hardware IFT is that registers, memory locations, and other hardware state are given a security label in addition to their functional value. The IFT model gives rules on how to update these labels as the hardware executes. And the IFT verification tools provide analysis techniques to understand if, how, and when information flows occur through the hardware. A crucial step in the hardware security verification process is formally describing the security properties, i.e., defining the threat model. Property-driven hardware security [7] requires the verification engineer to specify the important storage locations or assets. The IFT tools verify how the information contained in an asset moves throughout the hardware.

Consider a common hardware security verification example – understanding how a confidential asset, e.g, a secret key, can move throughout the hardware. In this case, the key register would be tagged and the IFT tools used to understand how, when, and where that information flows. Typically the verification engineer will specify where that information should not flow by providing a security boundary or a set of disallowed sink registers or memory locations. For example, the key information should not leak into user memory space. The IFT property may also involve conditions, e.g., the key cannot flow to the JTAG port except during debug mode. Properties related to integrity and timing can be crafted in a similar manner as described later in the chapter.

IFT verification involves understanding when, where, how, and why information flows occur. IFT verification techniques range from formal methods to simulation, emulation, and dynamic monitoring. Formal analysis provides guarantees on correctness and complete coverage but typically fails to scale past the level of IP cores. Simulation allows for larger analysis across a system on chip. Emulation allows for the even more complex analysis involving software and OS interactions. Dynamic monitoring performs real-time flow tracking.

This chapter serves to act as an introduction to the use of information flow tracking for hardware security verification. The aim is to provide the necessary background on information flow tracking models, properties, analysis, and verification tools, and then demonstrate how information flow tracking can be used in two case studies related to cache timing leakage and on-chip access control.

2 Information Flow

Information flow tracking (IFT) models how information propagates through a computing system. More specificallly, IFT provides the ability to label specific information in a system and understand how that information affects or moves throughout other parts of the system. The labels provide important meta-data about the system state that can be used to understand where information can leak (confidentiality), how data has been modified (integrity), and the ability to affect timing behaviors (availability). Thus, IFT is a fundamental security verification technique as it allows one to reason about properties related to confidentiality, integrity, and availability.

2.1 Information Flow Model

Dorothy E. Denning pioneered the idea of information flow tracking [8] defining an information flow model $\mathcal{IFT} = \langle \mathcal{N}, \mathcal{P}, \mathcal{SC}, \oplus, \rightarrow \rangle$ as a 5-tuple consisting of a set of storage objects $\mathcal{N}$, a set of processes $\mathcal{P}$, a set of security classes $\mathcal{SC}$, a class combining operator $\oplus$, and flow relation $\rightarrow$. IFT models are used in a variety of different abstraction levels including for hardware security verification [6]. This article focuses on IFT in the context of computer architecture, and therefore describes Denning's IFT model in that context. The *storage elements* consists of registers, memories, and other important architectural state. The *processes* are computations that the architecture performs, e.g., instructions, hardware accelerated functions, and interrupts. The *security classes* define how the storage objects can be labeled. When coupled with the *class combining operator*, security classes define how information flows as computation occurs. The *flow relation* defines the allowable flows between every pair of security classes. For example, one might first label registers that hold cryptographic key material as "secret" and label ports as "public," then further specify that information should never flow from "secret" to "public" elements. In this way, one can specify the flows that should and should not occur.

The security classes $\mathcal{SC}$ and their allowable flows $\rightarrow$ are key to using information flow tracking to verify security properties. Assume that there are two security classes: high and low ($\mathcal{SC} = \{H, L\}$) and a flow relation $L \rightarrow H$. The relation indicates that it is allowable for information labeled as L to flow to storage objects labeled as H. However, the opposite is not true: $H \nrightarrow L$; high information should never flow to storage objects with a L label. This simple lattice is shown in Figure 1a.

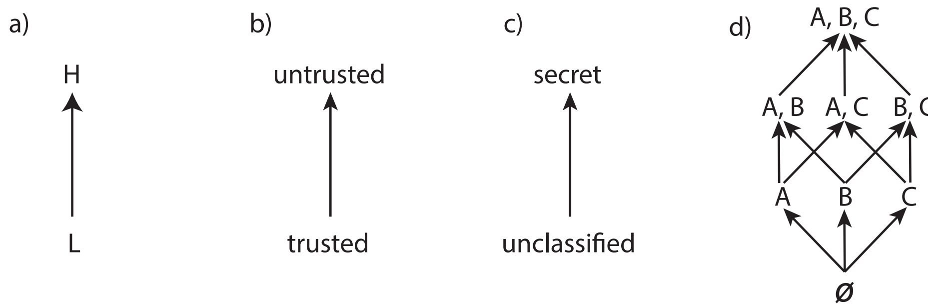


Figure 1: A security lattice defines the flow relationships $\rightarrow$ between the different labels of a security class $\mathcal{SC}$. Part a) defines a simple two label $\{H, L\}$ security class with allowable flow from low L to high H. This same lattice can be used to define integrity (Part b) and confidentiality (Part c) properties. More complex lattices are possible, e.g., Part d) shows a more complicated lattice that uses 8 different labels to indicate the mixing of information between three entities A, B and C. Most hardware security IFT tools use a simple two label lattice like those in Parts a, b, and c.

While simple, this two-label lattice is quite powerful. Viewing the lattice in light of integrity (Figure 1b), the H label would be considered *untrusted* and the L label is *trusted*. In this case, one wants to ensure that *untrusted* information can never affect a trusted storage object, i.e., *untrusted* $\nrightarrow$ *trusted*, which would violate the integrity of the system. One can also view this same lattice through the lens of confidentiality (Figure 1c) where *secret* information should never be leaked to an unclassified or openly-viewable storage object.

Lattices can also be more complex. Figure 1d shows a lattice with eight different security classes: $\{A, B, C\}$, $\{A, B\}$, $\{A, C\}$, $\{B, C\}$, $\{A\}$, $\{B\}$, $\{C\}$, and $\emptyset$. The flow relation defines how information from three different categories, A, B, and C, should be allowed to move throughout the system. A label of $\{A, B, C\}$ indicates that this storage object has information from all three categories, the label $\{A\}$ indicates the information only comes from A, the label $\{B, C\}$ denotes the object has information from both B and C, and so on. The flow relation depicted in this lattice says that information from security class $\{A\}$, for example, is allowed to flow to security class $\{A, B\}$ and transitively to $\{A, B, C\}$, but should never flow to security class $\{B, C\}$. Lattices can be arbitrarily complex, although in practice the vast majority of IFT tools default to a two-label lattice as shown in Figures 1a, 1b, and 1c.

A key idea of information flow tracking is that storage objects have a *security class label* in addition to their *functional value*. The label acts as an additional piece of metadata that IFT verification tools use to determine properties related to confidentiality, integrity, and availability.

IFT tools aim to provide some notion of tracking *noninterference* – an information flow model with a strict flow relation ($\rightarrow$) proposed by Goguen and Meseguer [9]. In the noninterference model of information flow any changes in high inputs shall never be reflected in the low outputs. That is, low objects can learn nothing about high information. Another way to think about this is that a noninterfering computer system should produce the same low outputs for any set of low inputs regardless of the functional values of the high inputs, i.e., it is impossible to learn any information about the high values by controlling the low inputs and viewing the low outputs. Equivalently, the computer system, projected on to a "low view" responds exactly the same to any input sequence of low values regardless of the high values.

IFT determines how the labels propagate throughout the system by analyzing the system behavior and updating the labels corresponding to the storage objects. The IFT tool is given an initial labeling for the storage objects. Setting these initial labels determines which objects are considered high and which are considered low, and therefore where information should and should not be allowed to flow. The rules for determining how an object's security class is updated are defined by the class combining operator $\oplus$.

IFT tools implement the class combining operators $\oplus$ to track information flows in different ways. The simplest and most conservative approach only considers the labels and marks the output of any process P as H when at least one of its inputs is H. In other words, the output of the process is labeled L only when all of the inputs to the process have an L label. In this case, the class combining operator is an OR-gate.[1] This is a safe approach, but can lead to false positives where data is labeled as H when it should be L, i.e., the IFT tool states there is a flow when one does not exist.

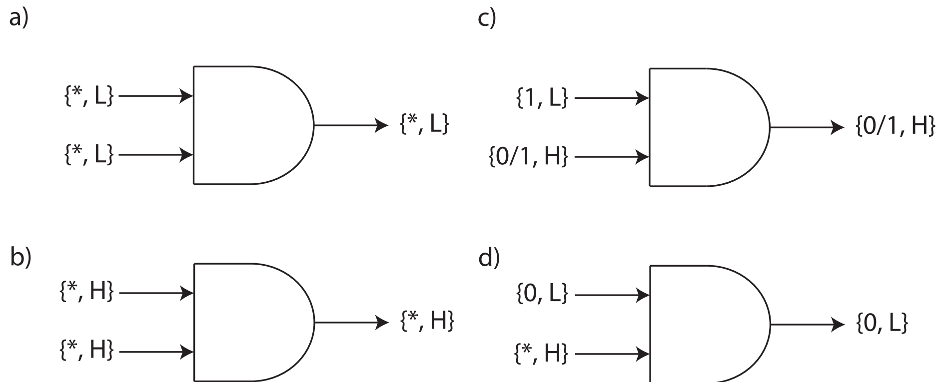


Figure 2: A simple process implementing a binary two-input Boolean AND operation. The two inputs have the functional values (0/1) and the corresponding security class labels (H/L). The output label is determined by the class combining operator. Part a) show an example when both inputs have an L label. The output should be labeled at L regardless of the functional values of the inputs (denoted as *). Part b) is similar where both inputs are H and the output label should be marked as H. Parts c) and d) show more complex examples when the input labels are mixed. Here the output labels depend on the functional values of the inputs. Part d) shows the specific scenario when an output can be labeled L even if one of the inputs has an H label.

The simple OR class combining operator is often too conservative and imprecise. It can quickly lead to many storage objects being marked as H even if they have no H information. To better understand the source of

[1] This logic assumes that the label $H = 1$ and $L = 0$.

this inprecision, consider a process that implements a binary two-input Boolean AND operation as shown in Figure 2. The inputs and outputs have their typical functional values (0 or 1). Additionally, each input/output has an associated label (H or L). The class combining operator is responsible for generating the output label. Part a) shows the case when both inputs have a L label. The output label will be L regardless of the input's functional value. Part b) illustrates a similar situation when the inputs all have an H label; the output will have an H label. More generally, if all inputs are H (L), the outputs will be H (L), respectively. The operator becomes more complex when one of the inputs is marked as H and the other input is L. Part c) shows a case when one of the inputs is labeled L and has a functional value of 1. The other input has a H label. The results of the AND operation will be equal to the value of H's functional value. Thus, there is information about H in the output – changing H will result in a direct interference of the output of the AND operation. Thus, the output should be labeled H. Part d) changes the functional value of the L input to 0. In this scenario, the output is always 0 regardless of the functional value of the H input. That is, no information related to the H input propagates to the output, thus the output can safely be labeled as L. Yet, the conservative approach would mark the output as H, i.e., stating that there is a flow from H to L when there is not. More complex combining operators consider the functional behavior of the process, the functional values of the inputs to that process, and their labels.

There are IFT models covering different types of flows – explicit and implicit [10], timing [11], and power [12]. This article focuses on explicit, implicit, and timing flows, which relate to the security properties that are more commonly verified in practice. The models differ on their security classes, class combining operator, and flow relations. Most of the variation is due to the class combining operator, e.g., determining explicit flows is generally much easier than determining implicit flows, which is generally easier than modeling timing flows. The basic ideas behind these different flows are described in the following. A hardware security verification engineer does not necessarily need to comprehend all the details how these flows are modeled – that is the job of the IFT tool – but they do require a basic understanding about the types of flows as those are important when specifying the information flow properties.

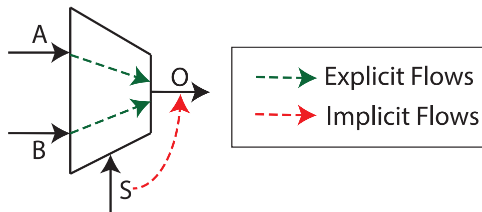


Figure 3: A multiplexor exhibits explicit information flows between inputs A and B and the output O. There is an implicit flow between the selector input S and the output O.

Figure 3 shows a simple example that illustrates the difference between *explicit flows* and *implicit flows*. The figure depicts a multiplexor as a process that has three inputs A, B, and S and one output O. Each input and output is associated with a storage object, e.g., a register/flip-flop. Explicit flows occur between the inputs A and B and the output O. In this case, the values from A or B are directly copied into O. This means that O contains exact information about A and B and thus O should take on the label of the input that is copied. The input S is the select bit which determines which of A or B to copy to the output O. There is an implicit flow between S and O due to the fact that an attacker would be able to determine information about the functional value of S by observing the functional values of O. Implicit information flows are more subtle, but are still capable of being exploited.

A *timing flow* is a scenario where information is transferred based upon the time that a process take to compute an output. A common timing flow occurs with caches. An attacker will request some data from the cache. The data is returned quickly if it is stored in the cache. When the data is not in the cache, it takes a longer amount of time to fetch it from memory. The actual value of the data in both cases is the same, but the time at which it is delivered is different. Thus, if that presence/absence of that data in the cache was affected by another H process, e.g., the H process loads data that evicts some other data, then the attacker can ascertain H information. This is the core idea between Spectre [13], Meltdown [14], Foreshadow [15] and other attacks that use the timing of cache operations to extract secret information.

2.2 Specifying Information Flow Properties

Information flow properties are specified by setting storage object labels and determining how those labels can and cannot flow throughout the system. Hardware security verification starts by developing a threat model and

determining the security assets to protect [16]. *Assets* are important system information whose behaviors should be monitored. A cryptographic key is a prime example of a security asset; a security verification engineer would want to understand how, when, and where information related to the key can move throughout the system. This is an example of a confidentiality property. Other examples of assets are control registers. Control registers often dictate important security scenarios. For example, a control register would be set in order to move the system into secure operating mode. Another control register would be set to indicate debug mode. Understanding who can set these control registers and the conditions under which they can be changed is important for the secure operation of the system. In this integrity scenario, one would want to understand when the trusted registers could be influenced by some untrusted storage object.

To better understand how to use IFT for hardware security verification, consider the control/status register (CSR) associated with setting secure operating mode (e.g., the Secure Configuration Register from TrustZone). This register stores important information related to operating in a secure mode and is use to move into and out of secure operating mode. Thus, it is important to protect the integrity of this register. In this scenario one would mark common registers and memory space corresponding to the non-secure world as untrusted and the CSR as trusted. One would want to determine if it is possible for the CSR label to ever become untrusted, and if it is, to understand the scenarios in which this can occur. IFT verification tools provide this ability.

IFT can also be used to understand security properties related to confidentiality. Consider the case where one aims to understand the confidentiality of a cryptographic key. IFT enables reasoning about how the value of a cryptographic key can flow throughout a computer system. In this scenario, assume the key is stored in a memory-mapped register associated with the custom IP core that performs the cryptographic operations. The system would be initially labeled in the following manner: the register that holds the key is labeled *secret* and everything else in the system is labeled *unclassified*. IFT tools can help determine which parts of the system can learn information about the key. For example, can information related to the key ever flow to the cache? IFT verification would taint the key and attempt to ascertain the conditions under which a storage object related to the cache becomes labeled as *secret*, i.e., information related to the key has been leaked into the cache. Additionally, verification engineers often wish to make strict "no flow" conditions. For example, no information related to the key should ever leak into user space. Here one would taint the key and use IFT to see if the physical memory locations corresponding to the user space can ever be marked as *secret*. If they are, then there is some information about the key in those tainted memory locations.

More generally speaking, labels are assigned depending on the security properties under verification. The labels can be broadly interpreted to define properties related to confidentiality, integrity, and availability. At their most abstract, labels are defined by the security classes. The rules for calculating the labels are defined by the class combining operator. When combined with the flow relations, these define the types of security properties that IFT can verify.

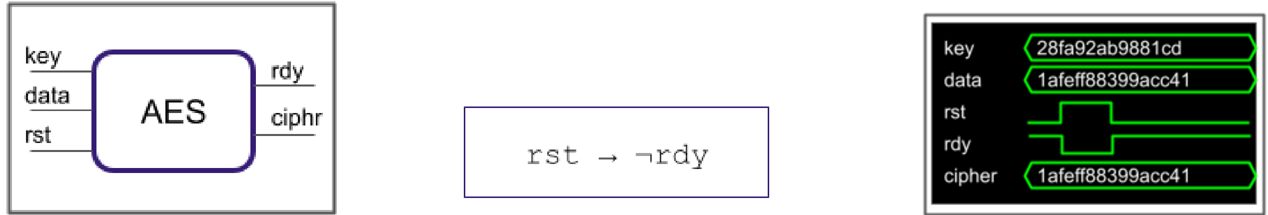
3 Information Flow Analysis

Analyzing how information flows through a design is important for security; however, classical analysis and verification techniques cannot be used to assess information flow properties. To see why this is, consider the simple design of an AES block cipher shown in Fig. 4a. The AES module takes as input a cryptographic key (`key`) and the data to be encrypted (`data`) and outputs the resulting ciphertext (`ciphr`). A ready signal (`rdy`) indicates when the ciphertext is valid and can be read. The ciphertext output is never valid while the module is undergoing a reset cycle, and the ready signal should reflect this. To test that the behavior of the ready signal is correct in this regard, one might write the simple assertion, $\texttt{rst} \rightarrow \neg\texttt{rdy}$ (Fig. 4b). Either traces of execution or a formal model of the design can then be studied to see whether it is indeed the case that the ready signal is never high during a reset cycle (Fig. 4c). Any trace-based verification technique can be used to either find violations of this property or determine that the behavior of `rdy` is likely correct because no violations are found.[2] Alternatively, a formal verification technique such as model checking can be used to either find violations of the property or determine that no violations can occur within the first N number of clock cycles of execution.[3]

Another desirable property of the module is that information about the key should not flow to the ready signal. It should be impossible to recover even one bit of the `key` signal by studying the on–off behavior of the `rdy` signal. A variation in time-to-compute can reveal information about the value of `key` when that variation depends on the value of `key`. The behavior of `rdy` will reflect any such variation and therefore will reveal information about

[2]Because exhaustive state coverage is not feasible, a trace-based technique will not *prove* correctness with respect to a property for any but the simplest of designs. The most such a technique can verify is that no violations were found.

[3]The bound N is an adjustable parameter to the verification tool.



(a) A simple AES module with inputs `key`, `data`, and `rst` and output `rdy`.

(b) An assertion to capture correct reset behavior.

(c) A trace of execution demonstrate the desired behavior.

Figure 4: A trace of execution can be used to confirm a simple assertion.

the key. When that happens, the `rdy` signal is said to *leak* information about `key`.

However, this property cannot be stated as a simple assertion in propositional logic. Let's say one wants to ensure that the value of the 0th bit of the `key` signal can never flow to the `rdy` signal. Two naively written properties might be $\texttt{key}_0 \rightarrow \texttt{rdy}$ or $\texttt{key}_0 \rightarrow \neg\texttt{rdy}$. However, neither one expresses the desired property, and in fact, both reflect acceptable behaviors of the design. It might be tempting to think that adding temporal logic to the property will solve the problem. For example, a property along the lines of $\texttt{key}_0 \rightarrow X(\texttt{rdy})$, which says that if the 0th bit of `key` is set then in the next (X) clock cycle `rdy` must also be set, seems to solve one problem with the naive properties, which is that information will surely take time to flow through a design. But, it does not solve the deeper problem, which is that there is no combination of values of $\texttt{key}_0$ and `rdy` that is illegal. Rather, it is that the value of `rdy` in a particular clock cycle should be the same whether `key` is set or unset; the value of `rdy` should not depend on the value of $\texttt{key}_0$. But, in order to get at this property it is not enough to reason about a single trace of execution. One needs to reason about traces in which $\texttt{key}_0$ is set *and* traces in which $\texttt{key}_0$ is unset. The property that is wanted is: in all possible traces, whenever all of the inputs other than $\texttt{key}_0$ are fixed, the behavior of `rdy` will be fixed. In other words, if `data`, `rst`, and $\texttt{key}_1 \ldots \texttt{key}_{n-1}$ are fixed to particular values, the waveform of `rdy` will not vary, regardless of how $\texttt{key}_0$ varies. This defines noninterference between $\texttt{key}_0$ and `rdy` – nothing about the value of $\texttt{key}_0$ can be learned by observing `rdy`.

First order logic (often abbreviated FOL) can be used to express the desired property that information should not flow from any of the bits of `key`. First order logic is more expressive than propositional logic and allows one to state a notion of *for all* traces. First order logic also allows one to introduce a notion of equality between two registers. The desired property, formally written, might look like this:

$$\forall\ \texttt{AES}_1, \texttt{AES}_2, \quad \texttt{rst}_1 = \texttt{rst}_2 \land \texttt{data}_1 = \texttt{data}_2 \rightarrow \texttt{rdy}_1 = \texttt{rdy}_2. \tag{1}$$

This property says that for any two traces produced by the AES module ($\forall\ \texttt{AES}_1, \texttt{AES}_2$), as long as `rst` has the same value in both traces and `data` has the same value in both traces, `rdy` will also have the same value in both traces.[4] The value of `key` does not affect the value of `rdy`. In other words, information does not flow from `key` to `rdy`. For simplicity, timing information has been elided; the true property would assert that the two `rdy` signals always have the same value at all points in the trace.

The key idea that makes the above property sound is that it reasons about *any* two possible traces. It is impossible to find two traces of the system in which `rst` and `data` are held fixed and the behavior of `rdy` varies.

3.1 Trace Properties and Hyperproperties

The example property about how information flows, written formally in Eqn. (1), is fundamentally different than the example assertion about how signals behave, shown in Fig. 4b. The latter is an example of a *trace property*. It is a property that can be exhibited by a single trace of execution, and can similarly be falsified by a single trace of execution. The types of properties that are expressible as SystemVerilog Assertions (SVA) are all trace properties. One usually thinks of these properties in terms of their logic formulation (e.g., $\texttt{rst} \rightarrow \neg\texttt{rdy}$), but another way to think about them is as a set of execution traces: for example, the set of all traces in which the statement $\texttt{rst} \rightarrow \neg\texttt{rdy}$ is valid. Using this definition in which a property is a set of traces, the property is true of a system if all of the traces the system could possibly produce are in the property's trace set.

[4] Note that one would probably like a stronger property saying that as long as `rst` carries the same value in both traces, regardless of both `data` and `key`, `rdy` will carry the same value in both traces.

The properties about how information flows, on the other hand, are not trace properties, but rather *hyperproperties* [17]. These properties are described by sets of traces, rather than by individual traces. Similarly, no single trace of execution can demonstrate a violation of a hyperproperty; only a set of two or more traces can do that. If a trace property is defined as a set of traces, then a hyperproperty is defined as a set of sets of traces, and every set of traces within the set represents a possible system satisfying the hyperproperty. Conversely, a hyperproperty is true of a system if all of the traces the system could possibly produce form one of the sets in the set of sets of traces that defines the hyperproperty. Hyperproperties can express notions of information flow. The AES hyperproperty is one example. Noninterference, described in Section 2.1 is another example, and determinism, which says that only the defined inputs can affect the output [18], is another. All of these information-flow hyperproperties are important for security. Hyperproperties can also express notions of fairness, such as whether a coin flip has a non-biased outcome. However, this chapter focuses on properties related to information flow.

Going forward, the term *property* will be used in a generic sense to mean the behavior of the system. Where the distinction is important and not clear from the context, the terms *trace property* and *hyperproperty* will be used.

3.2 Verifying Hyperproperties

A strong security verification effort will require verifying information flow properties − hyperproperties − of a design. However, because a hyperproperty is neither satisfied nor falsified by any single trace of execution, the traditional verification efforts will not work. It is not possible to express hyperproperties in standard assertion specification languages such as SVA. But, even if it were possible − if the specification language were updated to include the *forall* quantifiers, for example − the traditional verification approaches would not be able to determine whether such a hyperproperty is valid of the design or not. Trace-based engines monitor the signals of the design as simulation progresses and look for any violation of the given assertion. The engine does not have knowledge of any prior or future simulation runs, and therefore cannot reason about the comparative behavior of two or more traces of execution. Similarly, traditional model checking engines analyze the behavior of a single instance of the design and therefore cannot reason about the comparative behavior of two or more instances.

There are, however, new options for verifying information flow properties and they can be categorized by whether they use a static or dynamic analysis technique. Under the static analysis category are cone-of-influence analysis and a more sophisticated model checking technique (Section 3.2.1). Under the dynamic analysis category is information flow tracking (Section 3.2.2). Each of these are discussed in the following.

3.2.1 Static Analysis

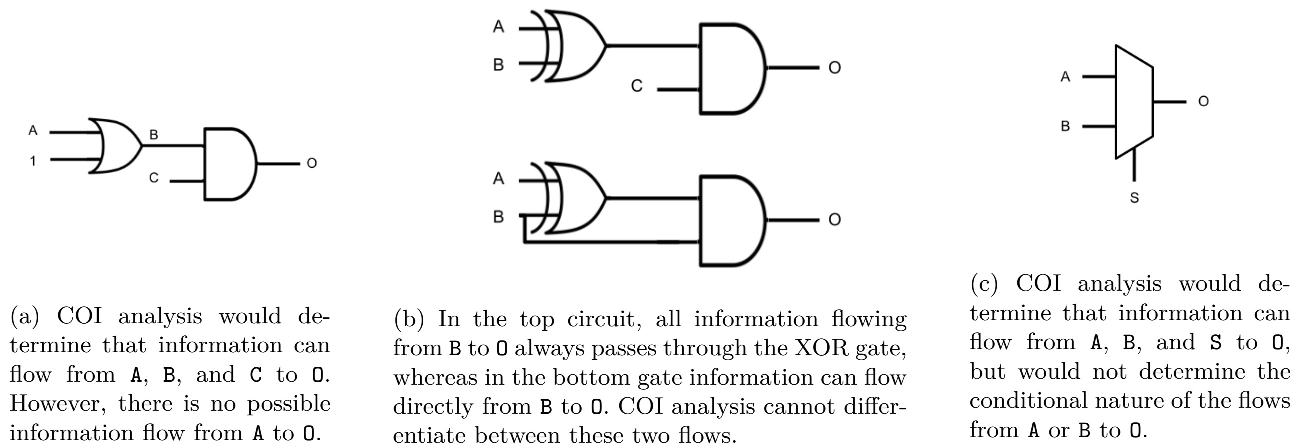
In static analysis the RTL description of the design is itself analyzed. The analysis tool takes as input the RTL design, but rather than try to simulate the design the analysis tool parses the design to answer a particular question.

Cone-of-Influence Analysis The simplest form of static analysis that can be used to verify information flow properties is a cone-of-influence analysis. This analysis finds every signal in the design that can possibly affect the behavior of a signal or set of signals, and it is often used in the course of classical traditional verification to simplify the verification problem. COI analysis can be used to provide some insight into how information flows as follows. Suppose there is a design D with output signal `snk` (for "sink"), and the verification goal is to identify which information flows to this sink signal. First, a *COI set* is initialized to include the signal `snk`. The analysis then identifies every signal s which appears on the right-hand-side of an assignment to `snk` in the RTL description of design D. Every newly identified s is added to the COI set. The analysis then repeats, for every s newly added to the COI set, every signal t which appears on the right-hand-side of an assignment to s is added to the COI set. The analysis continues to repeat until a steady-state is reached in which no new signals are added to the COI set. (As there are a finite number of signals in the design, the analysis is guaranteed to terminate.)

Every signal in the COI set has the potential to be a source of information flow to `snk`. This analysis is fast, requiring at most N iterations for a design with N unique signals, and the resulting COI set is complete: every signal which acts as a source of information flow to `snk` will be included in the COI set, or put another way, any signal that is not included in the COI set definitely does not influence the behavior of `snk`. However, the analysis is not sound: there may be signals included in the COI set which can never be a source of information flow to `snk`. Furthermore, the analysis provides a picture of how information flows in only broad strokes and

details about the path that information takes through the design or the conditions under which flows occur are not possible with this analysis.

To better understand the limitations of COI analysis, consider Figure 5. In Figure 5a, an OR gate tied to 1 is connected to an AND gate. It is clear that while information from B and C both flow to, and affect the behavior of, output O, no information is flowing from A to O since the value of B is determined solely by the fixed input. However, a COI analysis will not capture that fact and will include A in the COI set of O. Figure 5b illustrates how details about which path information takes will be lost by COI analysis. In the top circuit of Figure 5b, information flowing from B to O always passes through the XOR gate, while in the bottom circuit information can flow directly from B to O. COI analysis cannot make that distinction, and the distinction is important for security. Suppose A is a one-bit secret key and B is a one-bit message that should be kept private. If the key is generated at random and without bias and is unknown to the observer at O, then the observer cannot learn any information about the message B even though there is information flowing from B to O: XORing the private message with the secret key obscures the information in the message. However, in the bottom circuit, the observer at O will be able to learn information about B, for example, whenever the output at O is 1, the observer knows that the message is also 1. These two circuits have wildly different security postures, and in both cases the analysis if information flow is important, but a COI analysis cannot provide the needed information. Finally, Figure 5c illustrates a circuit with a conditional flow: depending on the value of S information will flow either from A or B to O. (Information always flows from S to O.) However, the COI analysis will put both A and B in the COI set and has no way to make note of the conditional nature of the flow.



(a) COI analysis would determine that information can flow from A, B, and C to O. However, there is no possible information flow from A to O.

(b) In the top circuit, all information flowing from B to O always passes through the XOR gate, whereas in the bottom gate information can flow directly from B to O. COI analysis cannot differentiate between these two flows.

(c) COI analysis would determine that information can flow from A, B, and S to O, but would not determine the conditional nature of the flows from A or B to O.

Figure 5: A trace of execution can be used to confirm a simple assertion.

Model Checking Model checking is a formal verification technique that provides complete security validation of a design. With model checking, a design and a desired property are expressed as a logical formula which is checked for validity. As a simple example, consider Figure 5a. The output of this circuit can be represented by the Boolean formula $(A \lor 1) \land C$. Let's suppose $C \rightarrow O$ is a desired property of the circuit. The model checking engine will search for a set of assignments to A and C that could violate this property. This is done by checking the satisfiability of the negation of the property: $\texttt{isSAT}(C \land \neg(A \lor 1 \land C))$. In this case, the SAT solver will be unable to find a satisfying solution to the formula, meaning the negation of the property cannot be satisfied and the desired property is true of the design. In reality the circuits to check are more complex, but the basic idea is the same: represent the circuit as a logical formula and check whether the negation of the property can be satisfied given the constraints of the circuit. If so, the satisfying solution serves as a counter-example to the desired property – a set of inputs to the design that will violate the property. On the other hand, if no satisfying solution is found, the property is true of the design.

Practical circuits include sequential as well as combinational logic. Techniques for sequential model checking include bounded model checking and k-induction, in which the design is unrolled so that a new logical formula representing the design is created for each clock cycle represented [19]; IC3 or property directed reachability, which does not require unrolling a design, but instead reasons about the reachability of a "bad" state from a given state [20, 21]; and engines based on binary decision diagrams (BDDs) which are used to efficiently represent sets of states and their transitions. The chapter on Bit-Level Model Checking describes the various techniques for sequential model checking in detail.

An out-of-the-box use of model checking in commercial verification engines cannot verify hyperproperties, which includes information flow properties. However, by carefully setting up the problem statement, model checking

can be used to verify some types of hyperproperties called *k-safety* properties. Information flow properties are k-safety properties, and in particular they are 2-safety properties. The way to use model checking to verify 2-safety properties is to use *self-composition*, a technique in which two identical instances of a design are combined in parallel to make one large design, which is then fed to the model checker [22]. Going back to the AES example from earlier, two instances of the design, say $\texttt{AES}_1$ and $\texttt{AES}_2$ are combined to create `AES`. The model checker can then verify the property $\texttt{rst}_1 = \texttt{rst}_2 \wedge \texttt{data}_1 = \texttt{data}_2 \rightarrow \texttt{rdy}_1 = \texttt{rdy}_2$ of this combined design. Model checking is expensive, the size and complexity of the satisfiability queries grow quickly and requiring two instances of a design doubles the starting complexity. For this reason, model checking information flow properties is limited to relatively small designs or to individual components of a design.

3.2.2 Dynamic Analysis

In dynamic analysis, it is the behavior of the design as it is simulated (or executed) that is analyzed. The analysis tool can be external to the design, in which case the tool can monitor only the input and output behaviors of the design. If, however, the behavior of internal signals needs to be analyzed then the design itself must first be *instrumented* – modified in such a way that the logic needed for analysis is incorporated into the design itself.

Trace-based analysis methods do not apply to information flow properties, and dynamic analysis is a trace-based method: the behavior of a trace of execution is observed and analyzed by the tool. However, by using information flow tracking, dynamic analysis can be made applicable to the study of information flow properties [23]. The design is instrumented to track how a particular input signal is affecting every other signal in the design. At the end of any single trace of execution, the added tracking logic has captured how information has flown from the input signal of interest.

To understand how this works consider the simple AND gate in Figure 6. In this example information flow tracking is used to expose how and when information from signal `B` can flow to output `O`. The original AND gate is on top and the added tracking logic, in this case a second AND gate, is on the bottom. The new signals $\texttt{B}_\texttt{T}$ and $\texttt{O}_\texttt{T}$ track the information as it flows through the circuit. Going back to the Denning model of information flow, $\texttt{B}_\texttt{T}$, $\texttt{O}_\texttt{T}$, and the second AND gate are implementing the class combining operator $\oplus$ of the underlying model. Information flows from `B` to `O` only when `A` is set, otherwise, the behavior of `O` is dominated by the unset `A` and unaffected by the behavior of `B`.

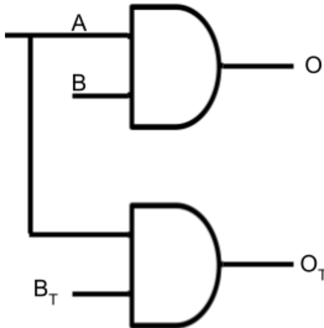


Figure 6: The tracking logic for a simple AND gate.

A design that has been instrumented with information flow tracking can then be run in simulation while the tracking signals are monitored to look for any undesirable flows of information. Verification engines automate much of the instrumentation and monitoring, allowing engineers to focus solely on specifying the desired properties and providing comprehensive testbenches. Returning to our AES example, verifying that no information flows from `key` to `rd` using information flow tracking would require writing the property `key` =/=> `rdy`, where =/=> is the not-flow operator used by many information-flow verification engines.

Information flow tracking does not suffer from the same issues of complexity that arise with model checking. On the other hand, it is a trace-based verification method and therefore cannot, in general, prove any property is true of a design, but rather can only find instances when the property is violated.

The following sections dive deeper into the strengths and limitations of information flow tracking and how to use it in practice.

4 Verification Tools

Modern verification tools allow engineers to verify a variety of properties related to information flow through the design. Commercially available tools offer advanced simulation-based and formal methods-based options for verifying information flow properties. In academia, new technologies are being developed, some of which will likely find their way into commercial tools down the line. This section presents and categorizes some of the current tools and discusses their strengths and weaknesses.

4.1 Simulation-Based Verification

The state of the art in simulation-based verification of information-flow properties uses information-flow tracking as described earlier. Source signals of interest are labeled and the design is instrumented to track how data propagates from the source signals throughout the rest of the design. The designer provides an information-flow property and testbench, the instrumented design is then simulated with the testbench providing input values, and the tool tracks whether the property is violated during the simulation run. An example property might state that information should flow from a particular source to a particular sink only when certain state conditions are met. The designer must provide a testbench that sufficiently exercises the design to find possible property violations. Simulation-based verification tools cannot prove the correctness of a design; if no property violation is found, it is possible that the testbench was insufficiently complete. On the other hand, if a property violation is found, the root-cause analysis is simplified as the testbench provides the exact sequence of inputs that caused the property violation.

One commercial tool using information flow tracking is Radix-S from Tortuga Logic [24]. The technology behind Tortuga Logic was first developed in academic research (for a survey, see [6]). Academic research has also demonstrated the use of information flow tracking to find timing channels in addition to data channels [25].

4.2 Formal Verification Methods

The state of the art in formal verification of information-flow properties uses a form of equivalence checking. The idea is similar in spirit to the use of self-composition with model checking described above and can be used to demonstrate determinism. The goal is to verify that a given destination signal is determined only by the known, allowed source signals. In other words, that there is no additional, illegal flow of information from a given source signal to a given destination signal. In equivalence checking, two versions of a design are proven to exhibit the same behavior given the same environment and inputs. To verify determinism, two copies of a design are created, and in both, the known sources of information are constrained to be equal. Sequential equivalence checking is done to verify that the destination signal in both copies will be equal. If it is possible for the two copies to diverge, then there exists some additional path of information flow to the destination.

A benefit to formal verification methods is that the result is not dependent on testbench coverage. If a violation is not found, then a violation does not exist. However, in order to handle large-scale designs, engineers often have to introduce suitable abstractions, and finding the right abstraction can be challenging. In addition, if a violation is found, performing the root-cause analysis can be difficult.

Commercial tools the perform formal verification of information-flow properties include JasperGold from Cadence [26], Questa Secure Check from Siemens [27], and Formal Security Verification from Synopsys [28].

5 Case Studies

This section presents two case studies for performing security verification of an hardware architecture description. The first case study performs security verification of a cache specifically targeting timing side channels. The second case study verifies memory access control systems – an important aspect of modern secure computing systems. In each case, the threat model is described, the assets are defined, and example security properties are presented.

5.1 Cache Timing Side Channels

Caches are crucial for high-performance computer architectures and present in all but the simplest microprocessors. Caches take advantage of spatial and temporal dependencies in data access patterns. When the processor makes a memory transaction, it assumes that data and its neighboring values will likely be accessed again in the near future, and thus keeps them in the faster, smaller cache memory. When a processor requests data that was already loaded into the cache, it is returned quickly (typically within a few cycles). When that data is

not in the cache, the cache itself must request it from another slower but larger memory (which typically takes tens of cycles) during which time the cache stalls. The variation in the time of retrieving data from results in a timing side channel [29]. This case study describes how to verify the existence of potential information leakage via a cache timing side channel. This leakage is powerful and a key element of Spectre [13], Meltdown [14], Foreshadow [15], and other architectural security attacks.

Figure 7 shows a diagram of a cache. The cache sits between a processor (left interface) and another memory (right interface). The cache stores a number of cache `data` lines each with metadata that includes their valid `v` bit, its corresponding memory `tag`, and its associated processor ID `pid`. The PID is used to differentiate between users, secure/insecure mode, etc. The `Processor Transceive Logic` takes as input write data `wr`, a read/write address `addr`, write (`wr_req`) and read request (`rd_req`) signals, and a process ID `pid`. The outputs are the processor's requested read data `rd` and the `stall` signal indicating whether the cache is waiting for data. The `Memory Transcieve Logic` interfaces with another larger and slower memory, e.g., a lower level cache or off-chip DRAM. This interface has an address `addr` and `rd` and `wr` busses that transfer cache lines between the cache and the memory. The terminology `processor.rd` and `memory.rd` are used when needed to disambiguate any unclear signal references.

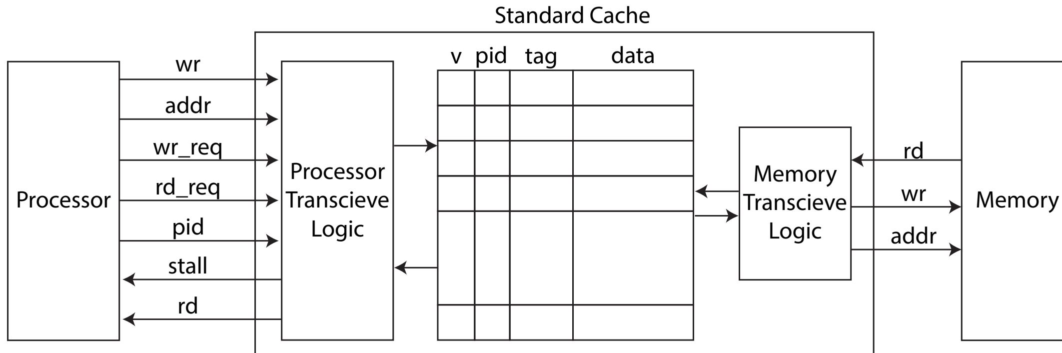


Figure 7: The case study aims to understand potential threats related to a cache timing side channel attack. The security verification focuses whether information can leak from a sensitive process (PID i) to another untrusted process (PID j) via the timing behavior of a cache. IFT can determine such complex interactions.

Assume that a threat model views a cache side channel as a security vulnerability. Thus, the security verification process aims to understand whether the cache is susceptible to an attack. The first responsibility is to identify the assets – what information needs protection, when is this information exposed to the cache, and where should it not flow? Specifying this in a manner that the IFT tool can understand involves interfacing with the security class labels. Digging further into the example will provide details on how and when to set and check the labels to verify a cache timing side channel.

Assume that the threat model requires that there is no timing side channel between PID i and PID j. Further, assume that PIDs are securely provided via the `pid` signal during processor read/write requests. One important cache side channel involves the leakage of information related to the memory accesses performed by some sensitive computation, in this case any computation performed by PID i. Thus, it is determined that `processor.addr` is an important asset that contains information that should not be leaked. `processor.addr` does not always carry information related to PID i; its label should only be marked high (H) when PID i is using the cache. Thus, the `addr_proc` label is set to H only if `pid == i`. The goal is to check when any sensitive address information about PID i flows from the cache to the processor while PID j is executing. `processor.rd` is one important signal where this information could flow. Thus, the analysis should assert that the `processor.rd` label is always L and report scenarios when it could become H as those are the conditions when information leakage about PID i's memory access pattern flows to PID j through a cache timing side channel. Finally, all storage objects (registers, Verilog variables, etc.) outside of the `addr_proc` signals should be initially marked as L.

IFT tools provide the capability to track different types of flows. In this case, the goal is to investigate timing flows, and thus, analysis requires an IFT tool that can track timing flows. Functional flows and timing flows differ in how the information is being transferred. *Functional flows* transfer information directly through the functional values of the storage object. Explicit flows are an example of a functional flow. *Timing flows* manifest themselves via the time at which the result is delivered.

In this cache timing side channel example, the goal is to know if the timing behavior of the cache provides any information to PID j about how PID i previously used the cache. A common timing flow manifests itself via the `processor.rd` signal, specifically the time at which valid data is presented on that signal. Note that there

may be a timing flow without a functional flow since the value of the `processor.rd` will not vary (only the time at which that same value is delivered). These ideas are formalized by Oberg et al. [11].

An IFT tool that tracks both functional flows and timing flows, e.g., Clepsydra [25], will provide separate labels and check the status of the functional (f) and timing labels (t). The notation used here appends the labels to storage objects. For example, `processor.addr.f` and `processor.addr.t` indicates the functional flow label and timing flow label (respectively) for storage object `processor.addr`. Functional flows and timing flows are not independent. A functional flow is transferred to a timing flow in the case(s) when that functional value is assigned conditionally. This causes a variation in the time at which that functional value is set, i.e., there is a timing flow.

Going back to the case study, the goal is to understand if the functional values `processor.addr` ever leak via a timing channel. Thus, the analysis requires setting `processor.addr.f = H` when PID i is using the cache, and checking whether the H label can ever flow to `processor.rd.t` (the timing label of the read data) when `pid == j`.

Listing 1 provides an assertion-based IFT property for determining if a cache exhibits a timing side channel via its access patterns. The assertions work directly on the labels and uses a labeling system similar to Clepsydra [25] for verifying functional and timing flows. The default functional `f` and timing `t` labels are set to L. The `processor.addr.f` functional label is set to H when PID i is using the cache, indicating that this information will be tracked. The analysis must then determine if any information about how PID i used the cache is leaked to `processor.rd` via a timing channel. This is reflected in the `processor.rd.t` value. If it is H, that indicates that `processor.rd` contains some information about `processor.addr` via a timing channel.

Listing 1: Assertion-based property covering cache timing side channel from Figure 7.

```
assume (default f,t == L);
if (pid == i)
        assume (processor.addr.f == H);
if (pid == j)
        assert (processor.rd.t == L)
```

All IFT tools provides a manner for setting and checking the IFT labels. That may be done directly, as shown above, or it could be through some higher level property specification. One common IFT property language feature is the "no flow" operator `=/=>` which indicates that information from a source storage object should not flow to a sink storage object, i.e., `source =/=> sink`. This is equivalent to setting the `source` label H and verifying that `sink` label stays L.

In addition to the `=/=>` operator, IFT tools often have a way to specify the conditions under information should be tracked (i.e., when to set the source label), and conditions under which flows are allowable (i.e., when to check if the `sink` label is H). Listing 2 cache side channel property using the no-flow operator:

Listing 2: No-flow property covering cache timing side channel from Figure 7.

```
processor.addr.f when (pid == i) =/=> processor.rd.t unless (pid == i)
```

This is equivalent to Listing 1, but provides a higher level abstraction for flow modeling. The `when` keyword provides the time to mark `processor.addr` as H and the `unless` keyword denotes that flow is allowable when `pid == i`, but at no other times.

Regardless of how the property is specified, a standard cache without any timing side channel mitigation should fail security verification related to a timing channel. That is, the verification process will find at least one scenario when information about PID i memory accesses are leaked to PID j via a timing channel on the `processor.rd` object. Without any mitigation in place, that scenario would occur when PID j accesses a cache line that was previously accessed by PID i.

5.2 Memory Access Control

Security-critical applications often have dynamic policies for securing memory transactions. Examples include: 1) a memory region is temporarily isolated when an untrusted application is executing, 2) kernel state is only accessible during debug mode, and 3) only the hardware root of trust can access and set security critical control and status registers.

In order to operate in a secure manner, processors use some form of an *access control system* that enforces an *access control policy*. The access control policy defines how different components of a processor (CPU core, custom accelerators, hardware root of trust, peripherals, memory management units, etc.) can access each other. The access control policy changes over the lifetime of the system – moving from the manufacturer, to the system integrators, and finally the end-users. Policies often change at runtime, e.g., different access restrictions occur during boot mode, secure operating modes, reset, debug, and normal operating modes.

The access control system plays a crucial role in maintaining a secure computing environments and must undergo a rigorous verification process to ensure that it is implemented correctly. Verification includes functional correctness. Additionally, and equally as important, the access control system must undergo a security verification that addresses potential security weaknesses and vulnerabilities. An exploit in the access control system endangers the confidentiality, integrity, and availability of the overall system.

Unfortunately, it is challenging to correctly implement access control systems. The MITRE common weakness enumeration (CWE) database reports a substantial and growing number of hardware weaknesses [30]. Restuccia et al. performed a systematic review and found 30 CWEs related to access control [31]. These weaknesses range from ensuring that configuration registers are properly reset, that interrupts are correctly handled, and that memory regions are isolated. The following provides more detail on some of these properties.

In order to better understand how to perform security verification, consider the on-chip access control module depicted in Figure 8. An access control wrapper is programmed by a trusted entity, e.g., a hardware root of trust, to restrict the memory accesses of a CPU core to other memory-addressable resources (different processors, custom accelerators, peripherals). The wrapper inspects the CPU core's read/write requests and only passes along requests that adhere to the access control manager configuration. A simple configuration would be to only allow address requests between a specified low address `BASE_LOW_ADDR` and high address `BASE_HIGH_ADDR`. If an illegal request is made, the access control wrapper takes an action, e.g., decoupling the IP core from the on-chip interconnect and sending an interrupt to the trusted entity. The access control wrapper uses standard AXI interfaces where a manager M initiates transactions and a subordinate S responds to the access requests.

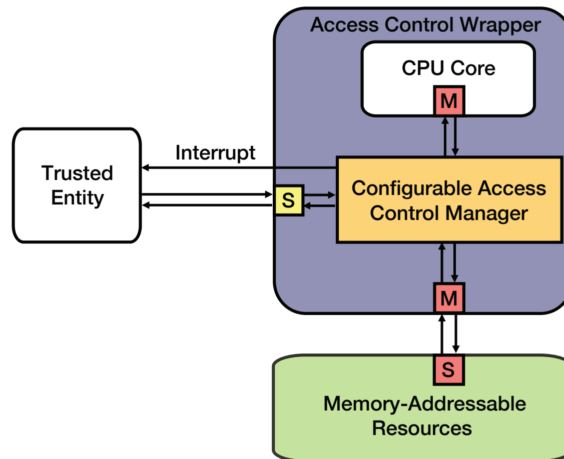


Figure 8: The access control wrapper inspects memory requests from the CPU core and relays them to the memory interconnect only if they adhere to the policy set in the Configurable Access Control Manager, which is programmed by a Trusted Entity. Any illegal accesses are stopped at the source and the trusted entity is alerted via an interrupt.

Security-critical properties of the access control module include both trace and hyperproperties. An example trace property is shown in Listing 3, which states that the AXI write address channel `AW` is always disabled after an interrupt occurs. This property can be gleaned by studying individual traces of execution. In each trace, whenever the `interrupt` signal is set, the `AW_CH_EN` signal will be unset. Identifying the pattern immediately gives rise to the property. It is also the case that the property can be disproved by a single trace of execution—one where both `interrupt` and `AW_CH_EN` are set.

Listing 3: An trace property that states the AXI Manager M address write channel is properly disabled during an interrupt.

```
interrupt |-> !AW_CH_EN
```

An example information flow hyperproperty is shown in Listing 4, which states that any information coming via

the CPU core's write port (`CPU.M.WDATA`) during an illegal write request[5] should not be reflected in the access control output write channel (`ACW.M.WDATA`).

Listing 4: An information flow property that states that information from the CPU core manager M interface should never flow outside of the access control wrapper when that information corresponds to an illegal request.

```
CPU.M.WDATA when illegal_request =/=> ACW.M.WDATA
```

Researchers have defined over 300 properties related to basic security behaviors of the access control wrapper; those properties along with the hardware description of the access control wrapper are available in the Aker open-source repository [32]. These properties were inspired by MITRE Common Weakness Enumerations (CWEs) [30] specifically those related to hardware design and include a mix of trace properties and hyperproperties.

Property generation was far and away the most challenging, time-consuming, yet important part of the security validation process. Automating property generation is invaluable in making security validation faster and more comprehensive, and new research has begun to do just that [33, 34, 35]. Potentially even more valuable is providing the engineer with insights into the design under validation, which could lead to the specification of additional properties and the discovery of new weaknesses and vulnerabilities.

6 Conclusion

This chapter demonstrates the benefits of a property-driven hardware security flow for verifying the security of hardware designs. Information flow tracking plays a crucial role in hardware security verification; information flow models and properties provide basic formalization of flow models and a language to specify security properties. IFT allows for the definition and verification of hyperproperties, which provide a way to reason about noninterference – an important aspect for verifying properties related to confidentiality and integrity. There are different tools and techniques for analyzing information flows each with their own strengths and weaknesses. The chapter concludes with two case studies demonstrating how to perform security verification for a cache timing side channel and on-chip access control.

References

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

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

[3] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 321–334, 2007.

[4] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres, "Making information flow explicit in histar," *Communications of the ACM*, vol. 54, no. 11, pp. 93–101, 2011.

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

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

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

[8] D. E. Denning, "A lattice model of secure information flow," 1976.

[9] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security & Privacy*, 1982, pp. 11–20.

[5] Indicated here by the pseudo signal `illegal_request`

[10] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

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

[12] A. Nahiyan, J. Park, M. He, Y. Iskander, F. Farahmandi, D. Forte, and M. Tehranipoor, "Script: a cad framework for power side-channel vulnerability assessment using information flow tracking and pattern generation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 25, no. 3, pp. 1–27, 2020.

[13] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[14] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[15] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 991–1008.

[16] S. Aftabjahani, R. Kastner, M. Tehranipoor, F. Farahmandi, J. Oberg, A. Nordstrom, N. Fern, and A. Althoff, "Special session: Cad for hardware security-automation is key to adoption of solutions," in *2021 IEEE 39th VLSI Test Symposium (VTS)*. IEEE, 2021, pp. 1–10.

[17] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, Sep. 2010, http://dl.acm.org/citation.cfm?id=1891823.1891830.

[18] A. Roscoe, "Csp and determinism in security modelling," in *Proceedings 1995 IEEE Symposium on Security and Privacy*, 1995, pp. 114–127.

[19] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," 2003.

[20] A. Bradley, "SAT-based model checking without unrolling," 2011.

[21] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, 2011, pp. 125–134.

[22] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Proceedings International Static Analysis Symposium*. Springer, 2005, pp. 352–367.

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

[24] "Radix-S," Tortuga Logic. [Online]. Available: https://tortugalogic.com/

[25] A. Ardeshiricham, W. Hu, and R. Kastner, "Clepsydra: Modeling timing flows in hardware designs," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 147–154.

[26] "JasperGold," Cadence. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[27] "Questa Secure Check," Siemens. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/secure-check/

[28] "Formal security verification," Synopsys. [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html

[29] C. Percival, "Cache missing for fun and profit," 2005.

[30] *The Common Weakness Enumeration Official Webpage*, MITRE, https://cwe.mitre.org/.

[31] F. Restuccia, A. Meza, and R. Kastner, “Aker: A design and verification framework for safe and secure soc access control,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2021.

[32] *Aker Github Repository*, https://github.com/KastnerRG/AKER-Access-Control.

[33] C. Deutschbein, A. Meza, F. Restuccia, R. Kastner, and C. Sturton, “Isadora: Automated information flow property generation for hardware designs,” in *Proceedings of the Workshop on Attacks and Solutions in Hardware Security (ASHES)*. ACM, 2021.

[34] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, “Identifying security critical properties for the dynamic verification of a processor,” in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017.

[35] C. Deutschbein, A. Meza, F. Restuccia, M. Gregoire, R. Kastner, and C. Sturton, “Toward hardware security property generation at scale,” *IEEE Security & Privacy*, May/June 2022, special issue: Formal Methods at Scale.