Leveraging Gate-Level Properties to Identify Hardware Timing Channels

Jason Oberg*, Sarah Meiklejohn*, Timothy Sherwood† and Ryan Kastner*
*Computer Science and Engineering, University of California, San Diego
{jkoberg,smeiklej,kastner}@cs.ucsd.edu
†Computer Science, University of California, Santa Barbara
sherwood@cs.ucsb.edu

***Abstract*—Modern embedded computing systems such as medical devices, airplanes, and automobiles continue to dominate some of the most critical aspects of our lives. In such systems, the movement of information throughout a device must be tightly controlled to prevent violations of privacy or integrity. Unfortunately, bounding the flow of information can often present a significant challenge, as information can flow through channels that are difficult to detect, such as timing channels. As has been demonstrated by recent research in hardware security, information flow tracking techniques deployed at the hardware or gate level show promise at identifying these "timing flows" but provide no formal statements about this claim nor mechanisms for separating out timing information from other types of flows.**

In this paper, we first prove that gate-level information flow tracking can in fact detect timing flows. In addition, we work to identify these timing flows separately from other flows by presenting a framework for identifying a different type of flow that we call *functional flows*. By using this framework to either confirm or rule out the existence of such flows, we leverage the previous work in hardware information flow tracking to effectively isolate timing flows. To show the effectiveness of this model, we demonstrate its usage on three practical examples: a shared bus (I^2C), a cache in a MIPS-based processor, and an RSA encryption core, all of which were written in Verilog/VHDL and then simulated in a variety of scenarios. In each scenario, we demonstrate how our framework can be used to identify timing and functional flows and also analyze our model's overhead.

***Index Terms*—Timing Channels, Hardware Security, Information Flow Tracking, Testing.**

I. Introduction

New research on hardware security has shown that it is possible to tightly constrain the flow of information in a system. With exploits being continuously exposed in many safety-critical embedded systems such as implantable medical devices [1] and automobiles [2], hardware security research is becoming increasingly sought after as a way to provide early detection and formal guarantees. Information flow tracking mechanisms found at the gate level [3], [4], [5], [6] have shown to be a promising solution to this class of security problems, as they allow designers to test security properties before a chip is ever fabricated.

The security problems that exist in hardware, more often than not, cannot be solved by software mechanisms alone. As an example, in order to show that two devices on a bus are non-intefering, it is required that the devices not intefere directly (i.e., by corrupt transmitted data) or through timing (i.e., by delays in response time). At first glance, these timing variations might seem benign, but they have been recently exploited by many to extract secret encryption keys from miss/hit delays in processor caches [7], [8], [9], [10]. These attacks rely on a exploiting information leaking through a *timing channel*, where an attacker is able to deduce information by simply measuring execution time. Since modern hardware is now increasingly coupled with non-determinism and hidden state, methods for detecting and even reasoning about these types of information leaks is an increasingly complex problem. In some cases, these timing channels might not fit within the threat model of the system and thus might not be of concern. Nevertheless, the techniques for assisting hardware designers in reasoning and understanding these types of leaks are a necessity. Only when hardware designers understand the potential leaks in their designs (including through time) can they make an informed decision about its security.

The properties found at the gate level, most notably using gate level information flow tracking (GLIFT), provide a promising remedy to this problem. Since GLIFT targets the lowest digital abstraction, it is able to detect and capture information leaking through time. This claim, however, is made in some of the initial work on GLIFT [3], [5], [11] but never thoroughly formalized. One of the specific contributions of this work is to make this formalism much more apparent and we do so in Section IV.

In addition, if a hardware designer using GLIFT detects that there is an information flow, there is no way to separate out the timing information from other *functional* information. Following the bus example as before, if a hardware designer were to observe an information flow using GLIFT, it would not be obvious whether or not this flow was from direct means (a device corrupting data on the bus) or by affecting another device's response time. Other timing-based information flows do happen quite frequently in modern computing systems as well. For example, in a system-on-chip (SoC) there may exist an access control mechanism between a core A and core B to prevent A from reading/writing to B. However, A issuing a READ/WRITE request to B may affect when B can respond to other requests. Thus A can affect the time in which core B can respond without directly affecting its data.

The second contribution of this work is to help solve this problem. We present a formal model in Section V that, when used in conjuction with GLIFT, isolates timing information from other flows of information. This model expands on our previous work [12] by providing more thorough and complete definitions, another application example (a shared bus), and more detailed discussion. As briefly mentioned, whether or not these timing flows are in the threat model depend on the system at hand. Nevertheless, this framework provides a way for hardware designers to *reason* about these timing flows.

To show the practicality of our framework, we explore in detail two common shared resources which are at the heart of interference in modern systems: the shared bus (Section VI) and CPU cache (Section VII). The shared bus in modern systems has been the source of the so called *bus-contention* channel [13] in which information can be covertly communicated through the traffic on a global bus. Previous work has explored how to identify information flows in global buses using GLIFT [11] but has fallen short of classifying these flows as functional or timing. Beyond the bus, we examine in Section VII the CPU cache; as previously mentioned, the cache is a common vulnerability in modern systems, as it is typically susceptible to leaking secret information through timing channels. As an additional data point, for a more thorough assessment of our technique, we demonstrate our analysis is effective at detecting timing channels in an RSA encryption core from opencores.org in Section VIII. For all examples, we do not make claims about complete information security, but rather increased confidence by identifying the presence of functional information and separating it from timing channels. Before we present our formal model and its use, we outline some essential preliminary definitions in Section III and formal definitions of GLIFT and information flow tracking using GLIFT in Section IV.

II. Threat Model

The specific threat model we target is hardware with potential timing channels that might adversely affect confidentiality or integrity. For confidentiality, we address the issue of designers being unable to determine whether or not an information leak is from timing or direct means. For example, caches have been of big concern when processes from different trust levels share cache lines. Data used by a secret program can be, and has been, extracted solely from the time it takes to perform memory operations.

For integrity and availability, we address concerns related to timing-based interference. For example, if a hardware designer is building a system-on-chip and wishes to isolate high-integrity cores from less trusted third-party ones, while still allowing resource sharing, then he could use this framework to reason about the timing effects that the less trusted cores have on the high-integrity ones. This type of property is often desired in the department of defense where Red-Black separation is required.

In both cases, our framework gives designers further insight into potential vulnerabilities so they can make better decisions. In some cases, these timing flows might be of no concern at all; i.e., the attack space of the cache or the timing effects on high-integrity cores are simply not in the threat model of the designer. Regardless, this work provides hardware designers with tools to more accurately evaluate their threat model, giving rise to increased confidence and more secure designs.

III. Preliminary Definitions

Before defining information flows and related concepts, we must first define some preliminary notions formally. Many of these notions are commonly understood by hardware designers, but we formulate them in such a way as to fit our model in a clear and concise manner. We start with the notion of time; as we are working at the gate level, the only notion of time that we consider is the system clock.

Definition 1. *We define the* clock *to be a function with no inputs that outputs values of the form $b \in \{0, 1\}$. We define a* clock tick *to be the event in which the output of the clock changes from* 0 *to* 1*. Finally, we define a* time t *to be the number of clock ticks that have occurred, and we define T to be the set containing all possible values of t.*

Our formal definition of time captures what we intuitively expect: some stateless hardware component will output a stream of ticks, and a separate stateful component will measure the number of ticks and use this to keep track of time. By keeping track of time, we can define an *event* as a given value at a certain point in time.

Definition 2. [14] *For a set of data values Y, a* discrete event *is the pair $e := (y, t)$ for $y \in Y$ and $t \in T$ (where we recall T is the set of all possible time values). We also define functions that recover the value and time components of an event as* $\mathsf{val}(e) = y$ *and* $\mathsf{time}(e) = t$ *respectively.*

To keep track of how values change over time, we can also define a sequence of events as a *trace*.

Definition 3. *For a value $n \in \mathbb{N}$ and a set Y, we define a* trace $A(Y, n)$ *to be a sequence of discrete events $\{e_i = (y_i, t_i)\}_{i=1}^{n}$ that is ordered by time; i.e.,* $\mathsf{time}(e_i) < \mathsf{time}(e_{i+1})$ *for all i, $1 \leq i < n$, and such that* $\mathsf{val}(e_i) \in Y$, $\mathsf{time}(e_i) \in T$ *for all i, $1 \leq i \leq n$. When the values of Y and n are clear, we omit them and refer to the trace simply as A.*

The way in which we have currently defined an event is quite broad: any value at any time can be considered an event. As an example, consider a system that outputs some value on every clock tick; if we run such a system for k clock ticks and record each output, then we will obtain a trace of size k. In many cases, however, events in this trace may be redundant, as the system might output the same value for many clock ticks while performing some computation. In this case, we would be interested not in the entire progression of events, but only in the case when the value of the output changes. To capture this, we define the *distinct* trace.

Definition 4. *For a trace $A(Y, n)$, we define the* distinct trace *of A to be the longest subsequence $d(A) \subseteq A(Y, n)$ such that for all $e_{i-1}, e_i \in d(A)$ it holds that $\mathsf{val}(e_i) \neq \mathsf{val}(e_{i-1})$.*

Constructing the distinct trace $d(A)$ of A is quite simple: first, include the first element of A in $d(A)$. Next, for each subsequent event e, check whether the last event e' in $d(A)$ is such that $\mathsf{val}(e') = \mathsf{val}(e)$; if this holds, then skip e (i.e., do not include it) and if it does not then add e to $d(A)$. As an example, consider a trace of two-bit values $A = ((00, 1), (00, 2), (01, 3), (01, 4), (11, 5), (10, 6))$. Then the distinct trace $d(A)$ will be $d(A) := ((00, 1), (01, 3), (11, 5), (10, 6))$, as the values at time 2 and 4 do not represent changes and will therefore be omitted.

With these definitions in hand, we can model a finite state machine system F that takes as input a value x in some set X and returns a value y in some set Y in a similar manner as past work [15]. To be fully general and consider systems that take in and output vectors rather than single elements, we assume that $X = X_1 \times \ldots \times X_n$ and that $Y = Y_1 \times \ldots \times Y_m$ for some $m, n \geq 1$, which means that an input x looks like $x = (x_1, \ldots, x_n)$ and an output y looks like $y = (y_1, \ldots, y_m)$. To furthermore acknowledge that the system is not static and thus both the inputs and outputs might change over time, we instead provide as input a trace $A(X, k)$ for some value k, and assume our output is a trace $A(Y, k)$.

Definition 5. [15] *A finite state machine (FSM) F is defined as $F = (X, Y, S, s_0, \delta, \alpha)$, where X is the set of inputs, Y the set of outputs, S the set of states, and $s_0 \in S$ the starting state. $\delta : X \times S \rightarrow S$ is the transfer function and $\alpha : X \times S \rightarrow Y$ is the output function.*

Since we are dealing with circuit implementations of finite state machines, both δ and α are represented as combinational logic functions. In addition, both δ and α can be called on a trace. $B = \alpha(A, s_0)$ generates a trace of output events $B = (e_0, e_1, \cdots e_k)$ during the execution on input trace A starting in state s_0. This notation describes α executing iteratively; it takes a state and trace as input and executes to completion producing an output trace. When the starting state is assumed to be the initial state, we use the notation $\alpha(A)$.

Now, since we are concerned with flows of information from a specific set of inputs (the subset of inputs which are of security concern), we need to formalize how to constrain the others. Recall first our intuition: an information flow exists for a set of inputs to the system F if their values affect the output (either the concrete value or its execution time). One natural way to then test whether or not these inputs affects the output is to change their value and see if the value of the output changes; concretely, this would mean running F on two different traces, in which the values of these inputs are different. In order to isolate just this set of inputs, however, it is necessary to keep the value of the other inputs the same. To ensure that this happens, we define what it means for two traces to be *value preserving*.

Definition 6. *For a set of inputs $\{x_i\}_{i \in I}$ and two traces $A(X, k) = (e_1, \ldots, e_k)$ and $A(X, k)' = (e'_1, \ldots, e'_k)$, we say the traces are* value preserving *with respect to I if for all $e_i \in A$ and $e'_i \in A'$ it is the case that $\mathsf{time}(e_i) = \mathsf{time}(e'_i)$, and if $\mathsf{val}(e_i) = (x_1, \ldots, x_n)$ and $\mathsf{val}(e'_i) = (x'_1, \ldots, x'_n)$, then $x_i = x'_i$ for all $i \notin I$.*

If two traces are value preserving, then by this definition we know that the only difference between them is the value of the *tainted* inputs $\{x_i\}_{i \in I}$. Taint will be formally defined shortly, but, as an example, secret data would be tainted and then tracked to ensure that it is not leaking to somewhere harmful. In this example, the set of secret inputs would be the set I. We will use this definition in the next section to prove that GLIFT detects both functional and timing information flows.

IV. Information Flow Tracking and GLIFT

Information flow tracking is a common method used in secure systems to ensure that secrecy and/or integrity of information is tightly controlled. Given a policy specifying the desired information flows, such as one requiring that secret information should not be observable by public objects, information flow tracking helps detect whether or not flows violating this policy are present.

In general, information flow tracking associates data with a label that specifies its security level and tracks how this label changes as the data *flows* through the system. As an example, consider a system with two labels: `public` and `secret`, and a policy that specifies that any data labeled as `secret` (e.g., a secret message) should not affect or flow to any data labeled as `public` (e.g., an untrusted shared memory) without first flowing through an encryption unit. More generally, information flow tracking can be extended to more complex policies and labeling systems (i.e., in general `high` data should never flow to `low`); as such, it has been used in all levels of the computing hierarchy, including programming languages [16], operating systems [17], and instruction-set/microarchitectures [18], [19]. Recently, information flow tracking was used by Tiwari et al. [3] at the level of logic gates in order to dynamically track the flows of each individual bit.

In the technique used by Tiwari et al., called gate level information flow tracking (GLIFT), the flow of information for individual bits is tracked as they propagate through Boolean gates; GLIFT was later used by Oberg et al. [11] to test for the absence of all information flows in the I^{2}C and USB bus protocols and by Tiwari et al. [4] to build a system that provably enforces strong non-interference. Further, it has been used to prove timing-based non-interference for a network-on-chip architecture in the research project SurfNoC [20]. Since its introduction, Tiwari et al. have expanded GLIFT to what they call “star-logic” which provides much stronger guarantees on information flow [4]. Briefly, GLIFT tracks flow through gates by associating with each data bit a one-bit label, commonly referred to as *taint*, and tracking this label using additional hardware known as *tracking logic*.

A. Formal definitions for GLIFT

To be precise, we present definitions of tracking logic and taint. First, it is important to understand how a "wire" in a logic function is tainted. We define this formally as follows:

Definition 7 (Taint). *For a set of wires (inputs, outputs, or internals) X, the corresponding* taint set *is X_t. A wire x_i for $x = (x_1, \ldots, x_i, \ldots, x_n) \in X$ is* tainted *by setting $x_{i_t} = 1$ for $x_t \in X_t$ and $x_t = (x_{1_t}, \ldots x_{i_t}, \ldots x_{n_t})$.*

In this definition, and in what follows, the elements of X and X_t are given as vectors; i.e., an element $x \in X$ has the form $x = (x_1, \ldots x_n)$ for $n \geq 1$. For single-bit security labels (which we use exclusively in this paper), $x \in X$ and its corresponding taint vector $x_t \in X_t$ are the same length.

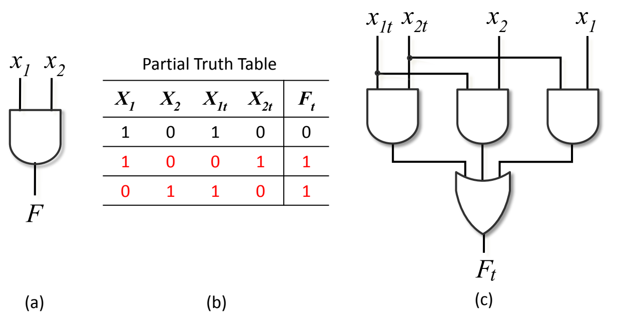
Now that we have a definition for taint, we can formally define the behavior of a tracking logic function and informatoin flow with a tracking logic function.

Definition 8 (Tracking logic). *For a combinational logic function $f : X \to Y$, the respective* tracking logic function *is $f_t : X_t \times X \to Y_t$, where X_t is the taint set of X and Y_t the taint set of Y. If $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$, then $f_t(x_1, \ldots, x_n, x_{1_t}, \ldots, x_{n_t}) = (y_{1_t}, \ldots, y_{m_t})$, where $y_{i_t} = 1$ indicates that some tainted input x_j (i.e., an input x_j such that $x_{j_t} = 1$) can affect the value of y_i.*

Definition 9 (Information flow). *For a combinational logic function $f : X \to Y$ and a set of inputs $\{x_i\}_{i \in I}$, an* information flow *exists with respect to an output y_j if $f_t(x_t) = (y_{1_t}, \ldots, y_{(j-1)_t}, 1, y_{(j+1)_t}, \ldots, y_{m_t})$, where each entry x_{i_t} of x_t is 1 if $i \in I$ and 0 otherwise. If there exists an index j such that $y_{j_t} = 1$, we just say an* information flow *exists.*

To understand how the tracking logic is used, consider a function with `public` and `secret` labels; then a label x_{i_t} is 1 if x_i is secret, and 0 otherwise. When considering a concrete assignment $(a_1, \ldots a_n)$ with each a_j being 0 or 1, running $f(a_1, \ldots, a_n)$ will produce the data output $(y_1, \ldots, y_i, \ldots, y_m)$, and running $f_t(a_1, \ldots, a_n, a_{1_t}, \ldots, a_{1_n})$ will indicate which tainted input can affect the values of which outputs (by outputting $y_{it} = 1$ if a tainted input affects the value of y_i and 0 otherwise). Going back to our sample function, if we observe some output $y_{i_t} = 1$ from f_t, we know that a secret input affects the output y_i of f. If y_i is public, then this flow would violate the security policy.

Typically, each individual gate and flip-flop is associated with such tracking logic in a compositional manner. In other words, for each individual gate (AND, OR, NAND, etc.), tracking logic is added which monitors the information flow through this particular gate. By composing the tracking logic for each gate and flip-flop together, we can form an entire hardware design consisting of all the original inputs and outputs, with the addition of security label inputs and outputs. Care must be taken to derive the tracking logic for each gate separately, however, as the way in which the inputs to a gate affect its output vary from gate to gate. As an example, consider the tracking logic for a AND gate as shown in Figure 1.



X_1	X_2	X_{1t}	X_{2t}	F_t
1	0	1	0	0
1	0	0	1	1
0	1	1	0	1

Fig. 1. (a) A simple AND gate. (b) A partial truth table for the tracking logic of an AND gate. $F_t = 1$ *iff* a tainted input affects F. (c) The tracking logic for an AND gate.

Simply by definition, we know that if some input of a AND gate is 0, the output will always be 0 regardless of the other inputs. In other words, if we have inputs $x_1 = 1$ and $x_2 = 0$ with security labels $x_{1_t} = 1$ and $x_{2_t} = 0$ as shown in Figure 1, then the output will actually be *untainted* even though $x_{1_t} = 1$, because the value of x_1 has no observable effect on the output of the gate (again, because $x_2 = 0$ and thus the output will be 1 regardless). By building a truth table for every gate primitive, tracking logic can be derived in this manner and stored in a library; the tracking logic can then be applied to the gate in a manner similar to technology mapping. As an example of how to compose these tracking logics, we consider a 2-input multiplexer (MUX), which is composed of two AND gates and a single OR gate where the output of the AND gates feed the inputs of the OR gate. First, the tracking logic for each AND gate and the single OR gate is generated. Then, the output of the tracking logic for each AND gate is fed as inputs to the tracking logic for the OR gate.

To use GLIFT in practice, a hardware description of the design is written in a hardware description language (HDL), such as Verilog or VHDL, and this description is then synthesized into a gate-level netlist using traditional synthesis tools such as Synopsys' Design Compiler. A gate-level netlist is a representation of the design completely in logic gates and flip-flops. Next, the GLIFT logic is added in a compositional manner (as we just described); i.e., for every gate in the system, we add associated tracking logic which takes as input the original gate inputs and their security labels and outputs a security label. Given a security policy such as our confidentiality example (i.e., secret inputs should not flow to the public output), GLIFT can then be used to ensure that the policy is not violated by checking that the output of the tracking logic f_t is not 1. It is important to remember that f_t is defined to report 1 *iff* a tainted input can actually *affect* the output. In other words, it will report 1 if at any instant in time a tainted input can affect the value of the output.

One of GLIFT's key properties is that it targets a very low level of computing abstraction; at such an abstraction, all information becomes explicit. In particular, because GLIFT tracks individual bits at this very low level, it can be used to explicitly identify timing channels. To support this claim, the following sections present some preliminary definitions and a model that, when used in conjunction with GLIFT, can test for timing channels. Such a model will be used in this paper to identify timing channels in a shared bus in Section VI, CPU cache in Section VII, and an RSA module in Section VIII.

B. GLIFT and timing channels

In order to have a clear understanding of timing channels, it first helps to specify a definition of a timing channel familiar to hardware designers. We define specifically a timing-only flow, where an input affects only the timestamp of output events and not the values. To be clear, we are concerned with timing leaks at the cycle level. Stated differently, we assume that an attacker does not have resources for measuring "glitches" within a combinational logic function itself. Rather, he can only observe timing variations in terms of number of cycles at register boundaries. With these assumptions, we present this definition in order to prove that GLIFT in fact captures such channels.

Definition 10 (Timing-only flow). *For a FSM F with input space X and output function α, a* timing-only flow *exists for a set of inputs $\{x_i\}_{i \in I}$ if there exists some value $k \in T$ and two input traces $A(X,k)$ and $A(X,k)'$ such that A and A' are value preserving with respect to I, and for $B = \alpha(A)$ and $B' = \alpha(A')$ it is the case that $\mathsf{val}(e_i) = \mathsf{val}(e_i')$ for all $e_i \in d(B)$ and $e_i' \in d(B')$ and there exist $e_j \in d(B)$ and $e_j' \in d(B')$ such that $\mathsf{time}(e_j) \neq \mathsf{time}(e_j')$.*

This definition captures the case in which a set of inputs affect only the *time* of the output. In other words, changing a subset of the tainted inputs will cause a change in the time in which the events appear on the output, but the values themselves remain the same. Before we can use this definition to prove that GLIFT captures timing-only channels, we need to define the GLIFT FSM F_t.

Referring back to Definition 5, a FSM consists of two combinational logic functions α and δ. Thus there exists tracking logic functions α_t and δ_t according to Definition 8. Using this property, we can define the GLIFT FSM F_t, which will be used to prove that GLIFT detects timing-only flows.

Definition 11. *Given a FSM $F = (X, Y, S, s_0, \delta, \alpha)$, the FSM tracking logic F_t is defined as $F_t = (X, X_t, Y_t, S, s_0, S_t, s_{0_t}, \delta_t, \alpha_t)$ where X, S, and s_0 are the same as in F, S_t is the set of tainted states, $s_{0_t} \in S_t$ the taint of the starting state, X_t is the set of tainted inputs, Y_t is the set of tainted outputs, δ_t the tracking logic of δ and α_t the tracking logic function of α.*

Now that these definitions are in place, we can prove that GLIFT can detect timing-only flows.

Theorem 1. *The FSM tracking logic F_t of a FSM F captures timing-only channels.*

Proof: Suppose there exists a timing-only channel for a finite state machine F with respect to the set of tainted inputs I. By Definition 10, this means there must exist value-preserving traces $A(X,k)$ and $A(X,k)'$ such that, for $B = \alpha(A)$ and $B' = \alpha(A')$, $\mathsf{val}(e_i) = \mathsf{val}(e_i')$ for all $e_i \in d(B)$ and $e_i' \in d(B)$, but there exist $e_j \in d(B)$ and $e_j' \in d(B')$ such that $\mathsf{time}(e_j) \neq \mathsf{time}(e_j')$. Since $e_j \in d(B)$ implies that $e_j \in B$ (and likewise for e_j'), this means that $B \neq B'$.

F generates an output every clock tick, so for all $e_j \in B$ and $e_j' \in B'$, $\mathsf{time}(e_j) = \mathsf{time}(e_j')$, and thus there must exist some $e_\ell \in B$ and $e_\ell' \in B'$ such that $\mathsf{val}(e_\ell) \neq \mathsf{val}(e_\ell')$ (because $B \neq B'$). By Definition 6, all input values remain the same for all $i \notin I$, meaning the only difference between them is in the tainted inputs, and thus the difference in output must have been caused by a tainted input. By Definition 8, α_t would thus have an output of $(y_{1_t}, \ldots, y_{\ell_t} = 1, \ldots, y_{m_t})$, as the value if y_ℓ in the output of α was affected by a tainted input. By Definition 9, this means GLIFT has indicated an information flow must exist. As the only possible flow is timing-based, GLIFT thus captures timing-only flows. ■

Since GLIFT operates at the lowest level of digital abstraction, all information flows become explicit. Thus, if at any instant in time a tainted input can affect the value of the output, GLIFT will indicate so by definition. At the FSM abstraction, as defined in Definition 10, this type of behavior often presents itself as a timing channel. This proof demonstrates that GLIFT can in fact identify these types of information flows. What is needed, however, is to formally understand how to separate these types of timing flows from other *functional* ones. In the next section we demonstrate how GLIFT can be used in conjuction with finding functional flows to isolate this timing information.

V. Isolating Timing Channels

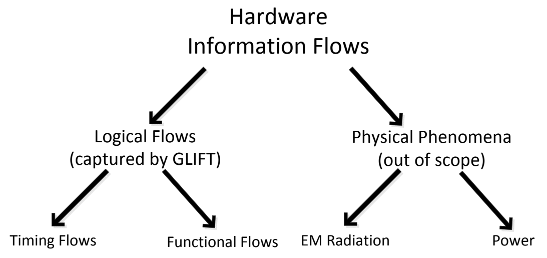

Hardware Information Flows
Logical Flows (captured by GLIFT)
Physical Phenomena (out of scope)
Timing Flows
Functional Flows
EM Radiation
Power

Fig. 2. The classes of information flows in hardware. In this work, we are concerned with logical flows that GLIFT captures, including timing and *functional* flows. Physical phenomena are out of the scope of this work.

As discussed in the previous section, GLIFT allows system designers to determine if any information flows exist within their systems even those through timing-channels. To be concise, at the digital level, there are two possible types of flows which we name *functional* flows and *timing*, as seen in Figure 2. Intuitively, a functional flow exists for a given set of inputs to a system if their values affects the values output by the system (for example, changing the value of a will affect the output of the function $f(a,b) := a+b$), while a timing flow exists if changes in the input affect how long the computation takes to execute.

While GLIFT will tell the designer only if any such flows exist, in this section we create a formal model for determining whether or not the system contains specifically functional flows. When used in conjunction with GLIFT, this technique therefore allows us to also determine what type of flow is occuring: if GLIFT determines that no flow exists, then clearly there is no flow. If instead GLIFT determines that a flow does exist but we can demonstrate that no functional flow exists, then we know that a timing flow must exist. What is left open, however, is the interesting case in which GLIFT determines that a flow exists but we determine that a functional flow does exist; in this case, we are unable to determine if a timing flow exists as well. In practice, however, benign functional flows are quite rare. Their biggest occurrences are in cryptographic operations (where the output is a direct function of the secret key) and in covert channels (where two subsystems will covertly communicate using varying amounts of seemingly functional noise). We would argue that in most other cases, a functional flow is likely to be something that violates a confidentiality or integrity policy. For example, a functional leak of the key without going through a cryptographic block would be detrimental to the security of the system.

A. Finding Functional Flows

Now that some intuition of the problem has been presented, we now discuss our testing framework as shown in Figure 3. Here GLIFT is used in conjuction with finding functional flows to isolate timing information. If GLIFT determines that there is no flow, we know there is no functional nor timing information flow. If, however, GLIFT determines there is a flow and we can find no functional flow with a reasonable number of traces, then we have increased confidence that the information flow occurred from a timing channel. In this section, we discuss how to find functional flows. We begin with the strongest possible definition and then weaken it to make it more amenable to testing techniques familiar to hardware designers.

Definition 12 (Functional flow)**.** *For a deterministic FSM F with input space X and output function α, we say that a* functional flow *exists with respect to a set of inputs $\{x_i\}_{i\in I}$ if there exists some value $k \in T$ and two input traces $A(X,k)$ and $A(X,k)'$ such that A and A' are value preserving with respect to I, and for $B := \alpha(A)$ and $B' := \alpha(A')$ it is the case that there exists $e_i \in d(B)$ and $e_i' \in d(B')$ such that* $\mathsf{val}(e_i) \neq \mathsf{val}(e_i')$.

This definition says that, if there is some functional flow from this set of inputs to the output, then there exist input traces of some size k that will demonstrate this flow; i.e., if a different output pattern is observed by changing only the values of these particular inputs, then their value does affect the value of the output and a functional flow must exist. In practice, however, this definition is not entirely useful: a

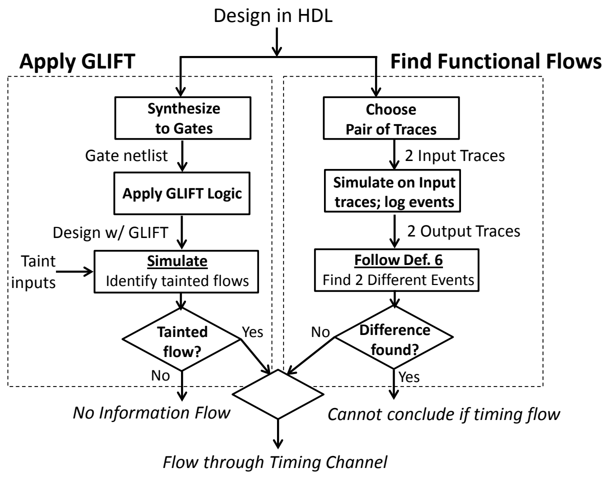


Fig. 3. How our method can be used with GLIFT to isolate timing channels. If GLIFT says there is a flow and we do not find a functional flow, we know there exists a timing channel. If we find a functional flow we cannot conclude the existence of a timing channel.

system designer wanting to isolate timing flows by ensuring that no functional flows exist would have to look, for every possible value of k, at every pair of traces of size k in which the value of this set of inputs differs in some way; only if he found no such pair for any value of k would he be able to conclude that no functional flow exists. We therefore consider how to meaningfully alter this definition so as to still provide some guarantees (albeit weaker ones) about the existence of functional flows, without requiring an exhaustive search (over a potentially infinite space!).

Definition 13 (Functional flow)**.** *For a deterministic FSM F with input space X and output function α, we say that a* functional flow *exists with respect to a set of inputs $\{x_i\}_{i\in I}$ and an input trace $A(X,k)$ if there exists an input trace $A(X,k)'$ such that A and A' are value preserving with respect to I and for $B := \alpha(A)$ and $B' := \alpha(A')$ it is the case that there exists $e_i \in d(B)$ and $e_i' \in d(B')$ such that* $\mathsf{val}(e_i) \neq \mathsf{val}(e_i')$.

At first glance, this definition already seems much more useful: instead of looking just at the set of inputs, we also consider fixing the first trace. If we then construct our second trace given this first trace to ensure that the two are value preserving, then comparing the distinct traces of the output will tell us if a functional flow exists for the trace. Once again, however, we must consider what a system designer would have to do to ensure that no functional flow exists: given the first trace A, he would have to construct all possible traces A'; if the distinct traces of the outputs were the same for all such A', then he could conclude that no functional flow existed with respect to A. Once again, this search space might be prohibitively large, so we consider one more meaningful weakening of the definition.

Definition 14 (Functional flow)**.** *For a deterministic FSM F with input space X and output function α, we say that a* functional flow *exists with respect to a set of inputs* $\{x_i\}_{i \in I}$ *and input traces* $A(X,k)$ *and* $A(X,k)'$ *that are value preserving with respect to* I *if for* $B := \alpha(A)$ *and* $B' := \alpha(A')$ *it is the case that there exists* $e_i \in d(B)$ *and* $e_i' \in d(B')$ *such that* $\mathsf{val}(e_i) \neq \mathsf{val}(e_i')$.

While this definition provides the weakest guarantees on the existence of a functional flow, it allows for the most efficient testing, as we need to pick only pairs of traces. Picking traces can be done in a variety of ways. The best approach is for the hardware designer to pick pairs of traces which will effectively stimulate the security issues in the designs. In general, however, this may be quite difficult since the person testing the hardware design may have limited knowledge of its operation. If the hardware designer has trouble picking two traces, a promising alternative is to pick random pairs of traces. In addition, the guarantees of this definition are not as weak as they might seem: they say that, given the output B, by observing B' as well, we are not learning any additional information about the inputs $\{x_i\}_{i \in I}$ than we learned just from seeing B. Again, while this does not imply the complete lack of any functional flow, it does provide evidence in that direction (and running this procedure with more, carefully chosen pairs of traces would only strengthen that evidence).

Finally, we discuss our requirement that the system F be deterministic, and observe that it is not as strict as it might seem. As discussed at the beginning of the section, we are interested only in flows that are detectable by GLIFT. Physical processes that can be used to generate randomness, such as the current power supply or electromagnetic radiation, are therefore out of the scope of this work. We can nevertheless consider randomness, however, in the form of something like a linear feedback shift register (LFSR), which is in fact deterministic given its current state; the randomness produced by an LFSR can therefore be held constant between two traces by using the same initial state.

B. A sample usage: fast/slow multiplier

To build intuition for how our model determines whether or not a functional flow exists, we consider a simple system as shown in Figure 4.

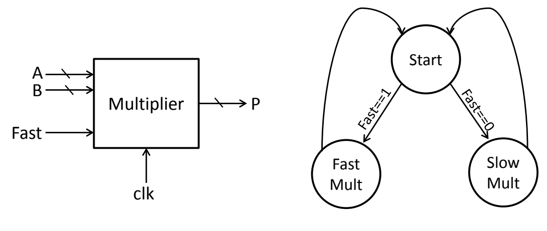


Fig. 4. On the left, we can see the inputs and outputs of the system S: it takes in two multi-bit inputs A and B and two single-bit inputs, fast and a clock input clk, and outputs $P := A \times B$. On the right, we can see that the system first picks an ALU to use based on the value of fast and then uses that ALU to perform the multiplication.

As we can see, the system consists of a pair of two-input multipliers, one fast and one slow. On inputs A, B, and fast, the system will use fast to determine which of the hardware multipliers to use. For both A and B, there is a clear functional flow from the input to the output, as $P := A \times B$. The input fast, however, has no effect on the value of the output P, as it simply selects whether to perform a fast or slow multiply. There is therefore no functional flow from fast to the output, but there is a clear timing flow, as we can see that the latency with which P is computed is highly dependent on the value of fast.

To confirm this intuition that the flow from fast must be timing rather than functional, we look at this input through the lens of our technique described above. Using as F the system in Figure 4, we can define the input space to be $X := (\mathbb{Z}, \mathbb{Z}, \{0,1\})$; i.e., all tuples consisting of two integer values and one bit, and our output space to be $Y := \mathbb{Z}$. As mentioned, we are interested in whether or not a functional flow exists for fast, so we will define this to be our set of inputs. Now, we pick values A_0 and B_0 for A and B respectively, and set our first trace to be $A := ((A_0, B_0, 0), t_0)$; i.e., the single event (at an arbitrary time t_0) in which A_0 and B_0 are multiplied using the slow ALU. We then set our second trace to be $A' := ((A_0, B_0, 1), t_0)$, and run these two traces to obtain output traces $B = (P, t)$ and $B' = (P', t')$. As A_0 and B_0 were the same for both traces, it is clearly the case that $P = P'$ and thus $\mathsf{val}(e_i) = \mathsf{val}(e_i')$ for all $e_i \in d(B)$ and $e_i' \in d(B')$, meaning no functional flow exists with respect to these two traces. As discussed above, this also provides evidence that no functional flow exists for fast at all, although further testing would likely be required to rule out this functional flow completely.

Although this example is a bit contrived, it effectively shows that finding hardware timing channels in practice is non-trivial, and testing for them requires some intuition (for example, knowing which traces to pick). In addition, many issues related to this method are analogous to those that may be encountered during conventional testing. For example, if a functional difference only manifests itself after N clock cycles and the hardware designer can only simulate for some number of cycles less than N, then he will not observe this difference. Some of these issues might be mitgated using some formal technologies, but we do not address those in this work and leave them for valuable future research. In the next section, we discuss a more complex example in which we examine how timing channels can be detected and eliminated in a shared bus system.

VI. The Bus Covert Channel

Shared buses, such as the inter-integrated circuit (I^{2}C) protocol, universal serial bus (USB), and ARM's system-on-chip AMBA bus, lie at the core of modern embedded applications. Buses and their protocols allow different hardware components to communicate with each other. For example, they are often used to configure functionality or offload work to co-processors (GPUs, DSPs, FPGAs, etc.). As the hardware in embedded systems continues to become more complex, so do the bus architectures themselves, which makes it non-trivial to spot potential security weaknesses in their construction.

In terms of such security weaknesses, a global bus that connects `high` and `low` entities has inherent security problems such as denial-of-service attacks, in which a malicious device can starve one of higher integrity, and bus-snooping, in which a low device can learn information from a high one. To ensure the terminology used here is well understood, we define a *timing side-channel* as an unintended leakage of information through how long a computation takes to run. A *timing covert-channel* (as used in this bus scenario) refers to an intended communication between two devices covertly by using variations in time.

The covert channels associated with common buses are well researched. One such channel, the *bus-contention channel* [13] arises when two devices on a shared bus communicate covertly by modulating the amount of observable traffic on the bus. For example, if a device A wishes to send information covertly to a device B, it can generate excessive traffic on the bus to transmit a 1 and minimal traffic to transmit a 0. Even if A is not permitted to directly exchange information with B, it still may transmit bits of information using this type of covert channel.

Both clock fuzzing [13] and probabilistic partitioning [21] have proven to be effective at reducing, if not eliminating, the bus-contention channel by inserting randomness into the system. They do not, however, expand beyond this particular channel and explore whether or not information might leak through other timing channels associated with the bus architecture. In addition, previous work using GLIFT has shown that strict information flow isolation can be obtained in a shared bus [11], but the work states nothing about how this information relates to timing. In what follows, we demonstrate how to use GLIFT and the techniques presented in Section V to prove that certain information flows in I^{2}C occur through timing channels.

A. Identifying Timing Flows in I^{2}C

The inter-integrated circuit (I^{2}C) protocol is a simple 2-wire bus protocol first proposed by Philips [22]. We chose to look specifically at I^{2}C because of both its wide usage in embedded applications for configuring peripherals and its simple structure; there is no reason, however, why the techniques presented here could not be applied to more sophisticated architectures or protocols.

In the I^{2}C protocol (seen in Figure 5), a "master" of the bus initiates a transaction by first sending a *start* bit by pulling down the data line (SDA) with the clock line (SCL) high. "Slaves" on the bus then listen for the master to indicate either a read or a write transaction. For write transactions, the master first sends a device address indicating a write and the device that matches this address responds with an acknowledgement (ACK). At this point, the master can transmit an internal register address (sub-address for the device) and the actual data. The transaction terminates with the master sending a *stop* bit. A similar behavior occurs for a read transaction, except here data transfers from a slave to the master. Since I^{2}C shares a common bus, there is the potential for several different covert channels, in addition to the bus-contention channel described above. To explore these different channels, we look at three configurations of the I^{2}C bus and discuss the potential ways in which information can be communicated covertly. We furthermore discuss how the flows in each of these covert communications can be classified as either a functional or timing flow using the techniques presented in Section V.

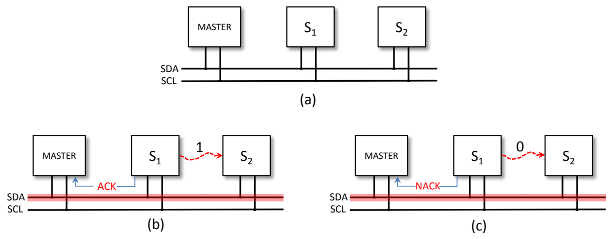


Fig. 5. (a) Standard I^{2}C configuration. (b) S_1 can covertly communicate a 1 to S_2 by sending an acknowledgement. (c) S_1 can communicate a 0 covertly to S_2 by sending a negative-acknowledgement.

1) Case 1: global bus: A global bus scenario, wherein multiple devices contend for a single bus, is the most general and commonly found bus configuration. Consider the example in which two devices wish to communicate covertly on the I^{2}C bus as shown in Figure 5. At first glance, there exists an obvious information flow in this architecture since the devices themselves can "snoop" the bus. For example, a device S_1 can send an acknowledgement to the master to covertly transmit a 1 to another device S_2; conversely, it can send a negative-acknowledgement to send a 0. Since S_2 observes all activity on the bus, it can simply monitor which type of message S_1 sends and thus determine the communicated bit. While this is by no means the only type of flow, for the sake of simplicity we will stick with this scenario throughout the rest of the section.

To put our model to use on this scenario, we designed the system shown in Figure 5 in Verilog by constructing I^{2}C Master and Slave controllers. Since we were interested in the flows between S_1 and S_2, we processed the designs in the manner presented in Section IV and in the previous work. To be concrete, we took the slave and master RTL descriptions and synthesized them down to logic gates using Synopsys' Design Compiler. For each gate primitive in the system, we added the appropriate GLIFT logic. The result is a system which contains a master and two slaves, each of which also has tracking logic associated with it. In a manner similar to that of previous work, we executed a test scenario wherein the master performs a write transaction with S_1 and S_1 sends an acknowledgement by simulating it in ModelSim 10.0a, a Verilog simulator. We observed that the GLIFT logic indicates a flow to S_2. At this stage, we have therefore identified that some type of information flow exists, but it is not entirely obvious if this was a functional or timing flow.

Since the devices can directly observe all interactions on the bus, one might expect this to be a functional flow. Not surprisingly, we utilized the model presented in Section V to show exactly that. To put this model to use, we abstract the output $y = \langle \mathrm{SCL}, \mathrm{SDA} \rangle$ of our model since these are the only two signals observable by S_2 (recall that SCL is the clock line and SDA the data line). In addition, we abstracted the input traces to our system as $A_1(X, k) := \langle S_1 \text{ sending NACK} \rangle$ and $A_2(X, k) := \langle S_1 \text{ sending ACK} \rangle$; running these through the system produced two output traces A_{G_1} and A_{G_2}. In a bit more detail, we collected A_{G_1} by logging the discrete events that occured when S_1 failed to acknowledge a write transaction from the master (thus intending to covertly transmit a 0). We then obtained a related trace A_{G_2}, in which S_1 does acknowledge the write. By analyzing these traces, we identified events $e_j \in d(A_{G_1})$ and $e'_j \in d(A_{G_2})$ (recall that $d(A_{G_1})$ and $d(A_{G_2})$ are the *distinct* traces of A_{G_1} and A_{G_2} respectively, as defined in Definition 4) such that $\mathsf{val}(e_j) \neq \mathsf{val}(e'_j)$. As a result, from Definition 14 of a functional flow, we know that a functional flow must exist. Recall, however, that this does not mean that there exists *only* a functional flow. Since GLIFT indicates that there exists a flow, it may be the case that information flows from S_1 to S_2 through both functional and timing channels.

The next case discusses how such a functional flow can be easily prevented using time-multiplexing of the bus in a manner similar to probabilistic partitioning [21].

2) Case 2: strict time-multiplexing of the bus: A seemingly easy solution to eliminate this information flow presented in Case 1 is to add strict partitioning between when devices may access the bus, as shown in Figure 6. Here, slaves on the bus may view the bus only within their designated time slots; this prevents devices from observing the bus traffic at all times. In this work, we partition over-conservatively by allowing the bus to be multiplexed between statically set time slots. In terms of probabilistic partitioning, we test the case in which the system is running in secure mode. We are interested in the same scenario as before: S_1 wishes to transmit information covertly with S_2; now, however, the bus-contention channel is eliminated, as partitioning has made contention impossible.

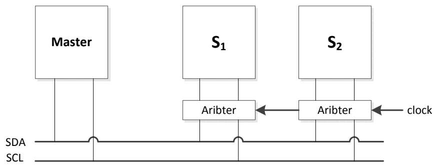


Fig. 6. Adding strict time-partitioning of the I^{2}C bus. The bus is only accessed by S_1 and S_2 in mutually-exclusive time slots.

Because the bus-contention channel has been ruled out, one might think that a covert channel between S_1 and S_2 no longer exists. Nevertheless, information can still be communicated covertly through the internal state of the master; to therefore transmit a covert bit, S_1 need only leave the master in a particular state before its time slot expires. For example, many bus protocols have a time-out period in case a device fails to respond to a request. If S_1 leaves the master in such a state prior to its time-slot expiring, S_2 can observe this state in the following time slot and conclude, based on the response time from the master, whether a 0 or a 1 is being transmitted: if the master's response time is short, S_2 can conclude S_1 wishes to communicate a 1, and if the response time is long it can conclude a 0. Although this type of covert channel is quite subtle, by using the model from Section V we can prove that this information flow occurs through a timing channel.

To prove that this is not a functional flow, we abstract this system in the same manner as Case 1, except we now use $y = \langle SDA_{S_2}, SCL_{S_2} \rangle$, where SDA_{S_2} and SCL_{S_2} are the wires observable by S_2. In the same manner as Case 1, we set input traces $A_1(X, k) := \langle S_1 \text{ sending NACK} \rangle$ and $A_2(X, k) := \langle S_1 \text{ sending ACK} \rangle$ to collect output traces A_{TDMA_1} and A_{TDMA_2} respectively. Following our model, we worked to find the existence of an event $e_j \in d(A_{TDMA_1})$ and $e'_j \in d(A_{TDMA_2})$ such that $\mathsf{val}(e_j) \neq \mathsf{val}(e'_j)$; we found, however, that no such events existed for this particular testing scenario. As discussed in Section V, this provides evidence for the absence of a functional flow; although it does not completely rule out the existence of such a flow, because we have chosen our input traces to represent essentially opposite events (sending a negative-acknowledgement and sending an acknowledgement), if a functional flow did exist then it is very likely it would be captured by these two traces. We therefore conclude that, because GLIFT did indicate the existence of some information flow and we have provided strong evidence that a functional flow does not exist, this flow is from a timing-channel.

3) Case 3: time-multiplexing with master reset: The work of Oberg et al. [11] using GLIFT for the I^{2}C channel indicated that all information flows are eliminated when the master device is reset back to a known state on the expiration of a slave's timeslot. In particular, this implies that no timing channels can exist, and thus the attack from Case 2 no longer applies. In practice, this trusted reset would need to come from a trusted entity such as a secure microkernel; we will therefore assume for our testing purposes that this reset comes from a reliable source once this subsystem is integrated into a larger system. With this assumption, we validated this scenario by adapting the test setup in Case 2 to incorporate the master being restored to an initial known state once S_1's time slot expires.

In the same manner as Case 2, we abstract the output $y = \langle SDA_{S_2}, SCL_{S_2} \rangle$. We create input traces $A_1(X, k) := \langle S_1 \text{ sending NACK} \rangle$ and $A_2(X, k) := \langle S_1 \text{ sending ACK} \rangle$ to log output traces A_{TDMA_1} and A_{TDMA_2} respectively. As expected, $d(A_{TDMA_1} = d(A_{TDMA_2})$, and thus we again obtain strong evidence that a functional flow does not exist.

As is hopefully demonstrated by these three cases, identifying the presented covert channels is not necessarily intuitive; furthermore, hardware designers are likely to easily overlook these problems when building their bus architectures or designing secure protocols. By combining the tracking logic of GLIFT with our model, we provide a method for hardware engineers to systematically evaluate their designs to determine whether or not techniques such as those used in Case 3 can in fact eliminate covert channels such as the ones presented in Case 1 and Case 2.

B. Overheads

To provide an understanding of the associated overheads with these techniques, we present the simulation times needed to execute them. We collected the simulation times by using ModelSim 10.0a and its built-in `time` function. The simulations were run on a machine running Windows 7 64-bit Professional with an Intel Core2 Quad CPU(Q9400) @ 2.66GHz and 4.0GB memory.

	Case 1	Case 2	Case 3
GLIFT	223.95 ms	230.29 ms	222.40 ms
RTL	210.45 ms	211.72 ms	219.04 ms

TABLE I
SIMULATION TIMES IN MILLISECONDS ASSOCIATED WITH THE THREE PRESENTED CASES FOR I^2C, AND FOR A SINGLE TRACE. GLIFT IMPOSES A SMALL OVERHEAD IN THE SIMULATION TIME FOR THESE TEST CASES.

As seen in Table I, there is not a significant difference between simulating the designs with GLIFT logic and the base register-transfer level (RTL) designs. This is likely due to the small size of the designs and the relatively short input traces required for these particular tests. The overheads associated with GLIFT become more apparent in Section VII when we discuss identifying timing channels associated with a CPU cache.

Finally, we mention that, although we consider two input traces for each case, we present in Table I our simulation times for only a single input trace. We do this because, as mentioned in Section V, designers may wish to check even beyond two traces to gain more assurance that a functional flow does not exist. Since the simulation time of a particular input trace is independent of the others, we chose to present the results for a single trace but note that they can be appropriately scaled to consider more traces as well.

VII. Cache Timing Channel

Recent work has shown CPU caches to be one of the biggest sources of hardware timing channels in modern processors [7], [8], [9], [10]. In a modern computing system, a cache can be seen as a performance optimization that provides a “quick look-up” for frequently used information. Caches are typically built from faster and higher power memory technologies, such as SRAM, and sit between slower main memory (typically DRAM) and the CPU core. When a memory region is referenced by a program, it is brought into the cache for fast access.

In previous work, the varying latencies of memory accesses due to cache hits/miss have been exploited. This vulnerability has been used to completely extract the secret key; these attacks have been divided into three categories: trace-driven [7], time-driven [8], [9], and access-driven [10]. Access-driven attacks in particular exploit knowledge about which cache lines are evicted. Specifically, a malicious process observes the latency of cache misses and hits and uses these patterns to deduce which cache lines are brought in/evicted, which in turn leaks information about the memory address (e.g., the secret key in AES table look-ups). In this work, we chose to look at access-driven attacks, as they are the easiest for us to demonstrate given our current test setup. Furthermore, this type of cache attack has applications beyond just encryption; for example, as demonstrated by Ristenpart et al. [23] in their attack on virtualized systems.

A. Overview of Access-Driven Timing Attacks

At a high-level, an access-driven cache timing attack first fills the cache using some malicious process. Next, a secret process uses a secret key to perform encryption. Finally, the malicious process tries to determine which of the cache lines were evicted in the encryption process. Since the key is XORed with part of the plaintext before indexing into a look-up table, the malicious process can correlate slow accesses with the value of the secret key.

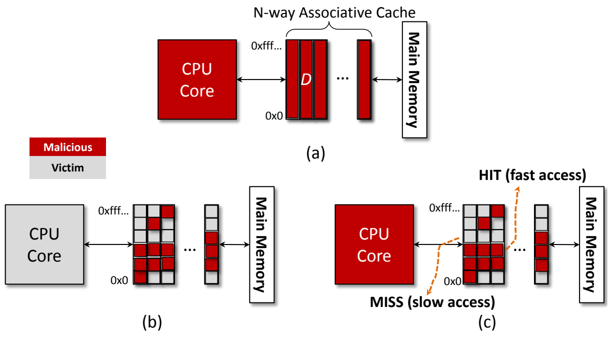


Fig. 7. (a) A typical CPU cache. The attack operates by malicious process first filling the cache with some data D. (b) The victim process encrypts some data with its secret key, thus bringing in cache lines. (c) The malicious process can observe which cache lines were evicted from latency, thus deducing the address and value of key used to index look-up table.

In a bit more detail, we can see a depiction of this attack in Figure 7. In our test setup, we have a malicious process M and secret process V (for victim). First, as seen in part (a), M fills the contents of the cache with some data D. Next, as seen in part (b), V subsequently runs AES using a secret key as input for a short duration; this process fills the contents of the cache. Now, in part (c), M reads D again from memory and observes the latency of each access. Since M and V share the cache, M will receive memory responses with lower latency if V did not evict certain cache lines prior to the context switch, as they will still reside in the cache. Because the secret key used by V is an index into look-up tables, the access latencies of M (i.e., a cache hit or miss) directly correlate with the value of the secret key.

B. Identifying the Cache Attack as a Timing Channel

Since this attack relies on the timing information available to M, it can clearly be identified as a type of timing attack. In this section, we demonstrate this fact more formally by using GLIFT and our model from Section V to prove that any information flows are timing-based.

To put this scenario to test, we designed a complete MIPS based processor written in Verilog. The processor is capable of running several of the SPEC 2006 [24] benchmarks including `mcf`, `specrand`, and `bzip2`, in addition to two security benchmarks: `sha` and `aes`, all of which are executed on the processor being simulated in ModelSim SE 10.0a (a commercial HDL simulator). All benchmarks are cross-compiled to the MIPS assembly using gcc and loaded into instruction memory using a Verilog testbench. The architecture of the processor consists of a 5-stage pipeline and 16K-entry direct mapped cache (1-way cache). We chose to use a direct-mapped cache for our experiments for ease of testing, but note that this analysis would apply directly to a cache with greater associativity.

Since our particular region of interest is the cache, we focus our analysis directly on this subsystem. To do so, we apply GLIFT logic to the cache system as described in Section IV. This new "GLIFTed" cache is re-inserted into the register-transfer level (RTL) processor design in the place of the original RTL cache. Pictorially, this can be seen in Figure 8. The input and output to the cache system include address and data lines and control signals (write-enable, memory stall signals, etc.); each such input and output is now associated with a taint bit which will be essential to testing whether or not information flows from our victim process V to our malicious process M.

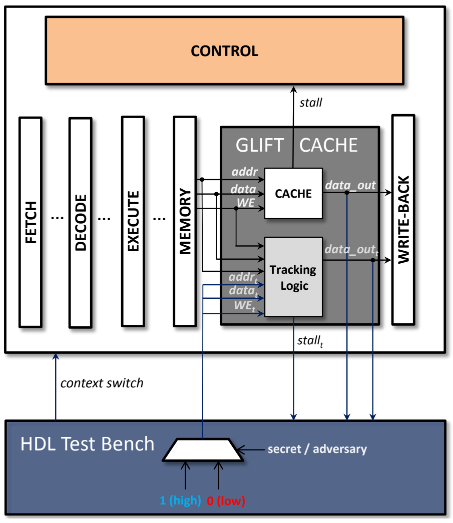


Fig. 8. A block diagram of a simple MIPS-based CPU. The cache is replaced by one that contains the original cache and its associated tracking logic. Our testbench drives the simulation of the processor to capture the output traces.

To execute the test scenario, we follow the same procedure as the access-driven timing attack previously discussed by having malicious and victim executions share the cache. We have M first fill the cache by setting all data in the cache. We then have V execute AES with all inputs to the cache marked as tainted (i.e. secret). Subsequently, we have M execute and observe whether or not information from V flows to M. As expected, we observe that as M reads from memory locations, secret information immediately flows out of the cache. We therefore know that a flow exists, but at this stage it is still ambiguous whether the flow is functional or timing.

To identify exactly which type of channel was identified by GLIFT, we leverage the benefits of our model by working to identify a functional flow; as previously discussed, if we detect no functional flow, then we know the flow must be from a timing channel. To fit our model, we abstract the output of the cache as $y = \langle data_M \rangle$ to indicate the cache output observable by M (note that, in particular, $stall$ is not included in this output, as it cannot be observed directly by M). Following our model, we then defined two traces: $A_1(X,k) := \langle V \text{ using } K_1 \rangle$ and $A_2(X,k) := \langle V \text{ using } K_2 \rangle$; i.e., the cases in which V encrypts using two different randomly selected keys. We then simulated both of these scenarios and logged all of the discrete events captured by ModelSim to obtain to output traces A_{C1} and A_{C2}; by definition of y, these output traces contain all events observable by M. Once we collected these traces, we checked whether or not a functional flow exists for these particular traces by looking for the existence of events $e_j \in d(A_{C1})$ and $e'_j \in d(A_{C2})$ such that $\mathsf{val}(e_j) \neq \mathsf{val}(e'_j)$. For these particular traces, we found no such pair of events. To build an intuition about why this type of information flow was not functional, it helps to look at the details of the cache structure. As mentioned, the cache in this scenario was designed to allow for a context switch between two running programs M and V. The cache control logic checks a process ID to ensure that M cannot access V's data and that only its own data is returned on a memory access. While V is running, all of its information in the cache is labeled as `secret`. When M executes, its memory accesses will not return V's data, but rather its own. If M's data was already present, this data would be returned quickly and if it were not then it would be returned with a delay in time. The same data would be returned regardless so there is no functional difference, rather only a difference in the *time* they appear. Again, although the fact that no functional flow exists with respect to these particular traces does not imply the lack of a functional flow for any traces, it does lend evidence to the theory that the flow must be timing-based rather than functional (and additional testing with different keys would provide further support).

C. Overheads

As we did for I^2C in Section VI, we evaluated the overheads associated with our technique by measuring simulation time. We collected our measurements using ModelSim 10.0a and its built in `time` function running on the same Windows 7 64-bit Professional machine with an Intel Core2 Quad CPU(Q9400) @ 2.66GHz and 4.0GB of memory. We measured the time for the secret process (V) to run AES on a secret key K_1 followed by a malicious process (M) attempting to observe which cache lines were evicted. This measurement was repeated for both the design with and without GLIFT. For completeness, we repeated the same process for the second input traces; namely when V executes AES using K_2 followed by M attempting to observe which cache lines were evicted. The resulting times from these simulations can be found in Table II.

	AES_{K_1}	AES_{K_2}
GLIFT	381.49 s	392.60 s
RTL	66.30 s	66.76 s

TABLE II
SIMULATION TIMES IN SECONDS FOR AES RUNNING WITH DIFFERENT ENCRYPTION KEYS, WITH AND WITHOUT GLIFT TRACKING LOGIC. IN GENERAL, SIMULATING A DESIGN WITH GLIFT LOGIC CAUSES LARGE SLOW-DOWNS.

As Table II shows, there is a large overhead ($\approx 6X$) for using GLIFT to detect whether or not a flow exists. Furthermore, since the behavior of M is fixed between both input traces and the only value changing is the secret key, the results clearly show that a timing channel exists with regards to the cache, as the execution time for AES on K_2 is longer than that of K_1; the existence of such a timing channel was also identified by GLIFT and our model.

VIII. TIMING CHANNELS IN RSA ENCRYPTION CORE

As an additional point of reference, this section describes how this model can be applied to detect a timing channel in an RSA cryptographic core. The RSA public-key cryptosystem [25] is one of the most widely used data encryption and digital signtuare algorithms. In short, the algorithm uses modular exponentiation to encrypt and decrypt data. Computing this exponentiation can be done quickly and efficiently in hardware.

One approach for computing decryption: $C^d \pmod{n}$, where C is the ciphertext, d is the private key and n the public modulus, is to employ a square-and-multiply algorithm which iterates over all key bits and performs a multiply each iteration depending on the value of the key bit. The details of this algorithm can be seen in Algorithm 1. If the current key-bit is 1, a multiply is performed otherwise the operation is skipped. A square is computed every iteration.

Algorithm 1 Basic algorithm for square-and-multiply to compute modular exponentiation. It computes $C^d \pmod{n}$.

```
$R = 1$;
$temp = C$;
for $i = 0$ to $|d| - 1$ do
    if bit $d[i] = 1$ then
        $R = R \cdot temp \pmod{n}$
    end if
    $temp = temp^2 \pmod{n}$
end for
return $R$
```

As one might expect, on iterations where an additional multiply is performed, the run time will be slower. Essentially, the value of the key will have great influence on the run-time of the decryption and thus attackers can (and have [26]) exploited this timing variation to extract the private key.

In hardware, an RSA decryption module[1] will not only have Key and Ciphertext inputs and a Message output, but other control signals as well. For example, a signal is needed to notify when the algorithm should begin ($start$) and also an output to say when the decryption is completed (rdy). If the key affects when the Message is ready (i.e. the time in which rdy is asserted), this timing variation can be exploited by an attacker.

A. Detecting Leak as Timing Channel

To this end, we apply our analysis to the BasicRSA core from opencores [27] and determine whether or not there is a timing channel in the design. Following the same GLIFT analysis flow, we detect that the key does in fact affect rdy. Now, to classify this as a timing leak, we apply the model presented in Section V and abstract the input traces $A := \langle \text{RSA on Key 1} \rangle$ and $A' := \langle \text{RSA on Key 2} \rangle$ using two randomly chosen keys and record the output traces B and B' by logging the values of the rdy signal for the duration of the decryption. When applying our model to these output traces, we find that $val(e_i) = val(e'_i)$ for all $e_i \in d(B)$ and $e'_i \in d(B')$. Since GLIFT indicates that there is an information leak and we did not detect a functional flow, we know that this leak must be from a timing-channel.

The analysis of this core brings up a necessary discussion. As described, our model cannot detect the presence of a timing when a functional one exists as well. For example, the ciphertext of an encryption algorithm (like RSA) will always be functionally affected by the key. However, as demonstrated here, by discovering the key's effect on the time in which rdy is asserted, it is possible to conclude that it affects the time in which the cryptographic process completes. In other words, this technique is able to conclude that the core has a timing channel.

IX. RELATED WORK

Most previous work on timing channels has focused on techniques for identifying timing and storage channels in larger systems, but not specifically in hardware design. Similarly, there has been significant work in reducing or eliminating specific timing channels, but little work in providing systematic testing techniques for identifying such channels.

Some of the most notable work in this area is with regards to the VAX Virtual Machine Monitor [28]. In one paper, Wray [29] describes how the timing and storage channels were analyzed in the VAX Virtual Machine Monitor; the timing channels described in his paper, however, are specific to the VAX VMM and a systematic testing method for identifying them was not discussed. In another paper, Kemmerer [30] presents a shared matrix methodology for identifying timing channels; this methodology works by creating a matrix that compares shared resources, processes, and resource attributes. Based on these fields and some proposed criteria for a timing and storage channel, the matrix can be analyzed to determine

[1]We use decryption here because RSA decrypts using a private key and encrypts with a public one

whether or not a shared resource can be used as a side channel. This technique therefore requires the designer to construct such a matrix and determine the shared resources, but ultimately still does not provide a general technique for detecting timing channels in hardware.

In terms of timing channel mitigation in secure systems, one technique (that we discussed in Section VI) is clock fuzzing, which was first introduced by Hu in 1991 [13]. Clock fuzzing works by presenting the system with a seemingly random clock to make it stochastically difficult for two objects to synchronize. However, as later discussed by Gray [21], clock fuzzing in reality only reduces the bandwidth of the timing channel and does not eliminate it entirely.

Recently, there has been extensive work with regards to hardware information flow tracking. Dynamic information flow tracking (DIFT), due to Suh et al. [18] tags information that comes from potentially untrusted channels and tracks them throughout a processor. This tag is checked before branches in execution are taken, and the branch is prevented if this information originated from an untrusted source. As demonstrated by Suh et al., DIFT is quite effective at detecting buffer overflow and format-string attacks, but works at too high of an abstraction to track information through timing channels. A similar tracking system, Minos [19], keeps an integrity bit on information and uses this bit to prevent potentially malicious branches in execution. Raksha [31] is a DIFT style processor that allows security policies to be reconfigured and thus provides a more flexible framework. As mentioned in Section IV, gate level information flow tracking (GLIFT) [3] works by tracking each individual bit in a hardware system. It is a general technique that has been applied to buid an execution lease CPU [5] and to analyze information flows in bus protocols [11]. Following this, some recent work from industry has shown that GLIFT-like techniques can be effectively applied in practice [32]. GLIFT itself precedes this work, but their use of similar methods shows that industry is searching for hardware security testing methods like the one presented in this paper.

Information flow tracking has also been used in hardware design languages. Caisson [33] is a hardware security language that aids hardware designers by using programming language type-based techniques to prevent unintended information flows and eliminate timing channels. This work is effective at helping hardware designers to build secure hardware, but is not a general technique for testing for timing channels. In this work, on the other hand, we have focused directly on a formal testing method for detecting hardware timing channels to make secure hardware easier to design and test.

X. Conclusions and Future Work

In this work, we presented a framework that can be used with gate-level information flow tracking (GLIFT) to effectively separate timing flows from functional flows. Using this separation, designers can make informed decisions about whether or not to be concerned with information flows identified by hardware information flow tracking techniques. In many cases, the designer is likely to be more concerned by timing channels than by functional flows, while in other cases the existence of timing channels might cause little concern.

To demonstrate the usefulness of our framework, we applied it to two common resources in modern systems: a shared bus and cache. In these examples, we showed how information flows can indeed be identified as timing-based with the help of gate level information flow tracking. While in some cases our framework does not provide any definite guarantees, it does provide strong evidence to rule out the existence of functional flows; used in combination with information flow tracking, which tells us if any flow exists, our framework can therefore provide strong evidence for the existence of timing channels.

Much future work is possible for both information flow tracking and for our framework in particular. Most prominently, if a functional flow exists then we cannot say anything about the existence of a timing flow; one natural question to ask is therefore if we can identify timing channels even in the presence of a functional flow. This would have implications for applications such as data encryption, in which the output ciphertext is always a function of the secret key, yet it is critical that an adversary observing encryption not be able to deduce the secret key using a timing channel. At the present, solving such a problem seems non-trivial and we leave it as an important open problem.

Another necessary future contribution is to more accurately evaluate the number of traces needed to detect a functional flow. At the gate-level abstraction, as discussed in this work, finding an answer to this issue appears to be a non-trivial problem. A potential remedy we have considered is performing our information flow analysis at an abstraction which does not include time. If we can treat parts of the system as atomic operations by eliminating the time component, then observed information flows would be purely functional. This approach may be a valuable asset to providing more formal guarantees about functional flows.

References

[1] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in *IEEE Symposium on Security and Privacy*, pp. 129 –142, 2008.

[2] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *Proceedings of IEEE Symposium on Security and Privacy ("Oakland") 2010*, pp. 447–462, 2010.

[3] M. Tiwari, H. Wassen, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of ASPLOS 2009*, 2009.

[4] M. Tiwari, J. Oberg, X. Li, J. Valamehr, T. E. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security," in *Proceedings of ISCA 2011*, pp. 189–200, 2011.

[5] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: a hardware-supported mechanism for enforcing strong non-interference," in *MICRO 2009*, MICRO 42, pp. 493–504, 2009.

[6] J. Oberg, T. Sherwood, and R. Kastner, "Eliminating timing information flows in a mix-trusted system-on-chip," *Design Test, IEEE*, vol. 30, no. 2, pp. 55–62, 2013.

[7] O. Aciiçmez and Çetin Kaya Koç, "Trace-driven cache attacks on AES (short paper)," in *ICICS*, pp. 112–121, 2006.

[8] D. J. Bernstein, "Cache-timing attacks on AES." Technical Report, 2005.

[9] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, pp. 1–20, 2006.

[10] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on AES to practice," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pp. 490–505, 2011.

[11] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I2C and USB," in *Proceedings of Design Automation Conference (DAC) 2011*, pp. 254 –259, 2011.

[12] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner, "A practical testing framework for isolating hardware timing channels," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 1281–1284, 2013.

[13] W.-M. Hu, "Reducing timing channels with fuzzy time," in *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pp. 8 –20, 1991.

[14] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, 1998.

[15] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, pp. 1090–1123, aug 1996.

[16] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, 2003.

[17] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *SOSP 2007*, pp. 321–334, 2007.

[18] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ASPLOS 2004*, pp. 85–96, 2004.

[19] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *MICRO 2004*, pp. 221–232, 2004.

[20] H. M. G. Wassel, Y. Gao, J. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "Surfnoc: a low latency and provably non-interfering approach to secure networks-on-chip.," in *ISCA*, pp. 583–594, ACM, 2013.

[21] J. W. Gray III, "On introducing noise into the bus-contention channel," in *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, pp. 90–98, 1993.

[22] "I2c manual." http://www.nxp.com/documents/ application_note/AN10216.pdf, March 2003.

[23] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds," in *Proceedings of CCS 2009*, pp. 199–212, 2009.

[24] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, pp. 1–17, 2006.

[25] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.

[26] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pp. 104–113, 1996.

[27] Opencores.org, "Basicrsa encryption engine." http://opencores.org/project,basicrsa, March 2009.

[28] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, "A retrospective on the VAX VMM security kernel," *IEEE Trans. Softw. Eng.*, pp. 1147–1165, 1991.

[29] J. C. Wray, "An analysis of covert timing channels," in *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pp. 2–7, 1991.

[30] R. A. Kemmerer, "Shared resource matrix methodology: an approach to identifying storage and timing channels," *ACM Trans. Comput. Syst.*, pp. 256–277, 1983.

[31] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *ISCA 2007*, pp. 482–493, 2007.

[32] D. Palmer and P. Manna, "An efficient algorithm for identifying security relevant logic and vulnerabilities in rtl designs," in *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, pp. 61–66, June 2013.

[33] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *PLDI 2011*, pp. 109–120, 2011.

Jason Oberg received his Ph.D. and M.S. degrees in Computer Science and Engineering from UC San Diego. He also received his Bachelor of Science degree in Computer Engineering from UC Santa Barbara in 2009. His research interests are primarily in hardware security with an emphasis on testing and verification methods for secure hardware and embedded system design. He is also a fellow of the National Science Foundation and Co-founder of Tortuga Logic, Inc., a hardware security company.

Sarah Meiklejohn is, as of September 2014, a Lecturer at University College London with broad research interests in cryptography and security. Previously, she obtained her PhD in Computer Science at UC San Diego, where she was co-advised by Mihir Bellare and Stefan Savage. Before that, she received and Sc.M. in Computer Science and an Sc.B. in Mathematics from Brown University.

Timothy Sherwood is a Professor of Computer Science at UC Santa Barbara and Senior Member of IEEE. His work has included the development of software profiling/debugging peripherals, novel uses of 3D integrated circuits for security and introspection, whiteboard-based sketch computing, MEMS sensor closed-loop control architectures, and high-assurance hardware/software systems. On 6 separate occasions he has had his papers selected by "IEEE Micro Top Picks", he is a recipient of an NSF Career Award, and the Northrup Grumman Excellence in Teaching Award. Prior to joining UCSB, he graduated with a B.S. in Computer Science and Engineering from UC Davis (1998), and received his M.S. and Ph.D. from UC San Diego (2003).

Ryan Kastner is currently a professor in the Department of Computer Science and Engineering at the University of California, San Diego. He received a PhD in Computer Science at UCLA, a masters degree (MS) in engineering and bachelor degrees (BS) in both Electrical Engineering and Computer Engineering, all from Northwestern University. He is the co-director of the Wireless Embedded Systems Master of Advanced Studies Program. He also co-directs the Engineers for Exploration Program. His current research interests reside in three areas: hardware acceleration, hardware security, and remote sensing.