Enforcing Information Flow Guarantees in Reconfigurable Systems with Mix-trusted IP

Ryan Kastner[‡], Jason Oberg[‡], Wei Hu[†,‡], Ali Irturk[‡]
[‡]Computer Science and Engineering, University of California, San Diego
[†]Automation, Northwestern Polytechnical University, Xi'an, China
{kastner,jkoberg,w3hu,airturk}@cs.ucsd.edu

***Abstract*—Trusted systems fundamentally rely on the ability to tightly control the flow of information both in-to and out-of the device. Due to their inherent programmability, reconfigurable systems are riddled with security holes (timing channels, undefined behaviors, storage channels, backdoors) which can be used as a foothold for attackers to strike. System designers are constantly forced to respond to these attacks, often only after significant damage has been inflicted. We propose to use the reconfigurable nature of the system to our advantage by taking a bottom-up, hardware based approach to security. Using an information flow secure hardware foundation, which can precisely verify all information flows from Boolean gates, security can be verified all the way up the system stack. This can be used to ensure private keys are never leaked (for secrecy), and that untrusted information will not be used in the making of critical decisions (for safety and fault tolerance).**

I. Introduction

Reconfigurable hardware frequently finds itself in charge of high-assurance applications such as flight control and medical systems. As these reconfigurable systems increase in design complexity, commercial off the shelf (COTS) intellectual property (IP) cores are becoming a commodity in these systems. Ensuring that the mix-trust in these systems does not violate the integrity or confidentiality of the system as a whole is required for its trusted operation. Such assurance often requires detailed verification including strict theorem proving and third party analysis [1]. This complicated process not only takes a tremendous amount of time estimated at 10 years [3] but also costs on the order of $10,000 per line of code [2]. Reducing this overhead is needed to keep these operation critical systems up-to-date with current technology at a reasonable cost.

An example of a system in which mix-trusted components interact can be found in a modern aircraft where a shared physical bus multiplexes between mix-trusted subsystems [8]. For example, if a bus arbitrates between user and flight control systems, unintended information flowing from the user to flight control system could have catastrophic consequences. It is absolutely required that untrusted information never corrupts trusted flight control components. If such connectivity is to be allowed, strict guarantees of information flow control are necessary to the correct and reliable operation of the aircraft.

To guarantee mix-trusted information flows only to where the system designer intends, information flow tracking (IFT) has been introduced. IFT works by monitoring the movement of data as it propagates through the system. Information flow tracking can be used to ensure that a secret key does not leak, in the context of Bell & LaPadula confidentiality [7] or to guarantee the integrity of trusted components, in the case of non-interference [6]. In general, there are two classes of information flows: *explicit* and *implicit*. Explicit information flows result from direct communication. For example, an explicit flow would occur between a host and device on a bus that were directly exchanging data or between processes over an inter-processes communication (IPC) mechanism. Implicit information flows are much more subtle and generally leak information through behavior. A common implicit information flow that occurs in hardware is a timing channel where information can be extracted from the latency of operations.

To account for these difficult to detect security holes, current methods are lacking in that they either perform physical isolation or "clock fuzzing" [14], [25]. Physical isolation works by separating trusted/untrusted or classified/unclassified subsystems from one another and allowing interconnect only at statically defined locations. This method is often effective although it tends to result in large area overheads since typical implementations require the replication of hardware. "Clock fuzzing" makes attempts to avoid physical isolation by presenting untrusted subsystems with a "fuzzed" clock that produces artificial errors in timing information. This tries to reduce the ability to gain information from timing channels but in reality only decreases the bandwidth of the channel. A more robust solution is to eliminate all such information flows and verify their absence; this can currently be done using a technique called gate level information flow tracking.

Gate Level Information Flow Tracking (GLIFT) [26] provides a solution for tracking information flows in hardware. Since GLIFT tracks information at the granularity of Boolean gates, it can be used on any digital hardware. Furthermore, it is capable of detecting logical flows including those through timing channels because all information becomes explicit in hardware. This does, however, focus only on logical flows and excludes physical phenomena such as power channels and EM radiation. This bottom-up approach to security can be seen in Figure 1 where a secure hardware foundation is created to ensure information flow security at the lowest abstraction level. With this secure hardware base, microarchitectural designs can be implemented with strict information flow guarantees.

This paper discusses the concerns with using mix-trusted IP in reconfigurable high-assurance systems and an overview

of current techniques for information flow control in them. It addresses how mix-trusted components can be interconnected with confidence that they will adhere to the defined information flow policy. It specifically focuses on this bottom-up approach to security by providing a secure computing base to build up from. It also discusses several areas of future research in the security in reconfigurable systems. This includes providing better use of untrusted IP and how to provide the greatest reconfigurability while maintaining strict security.

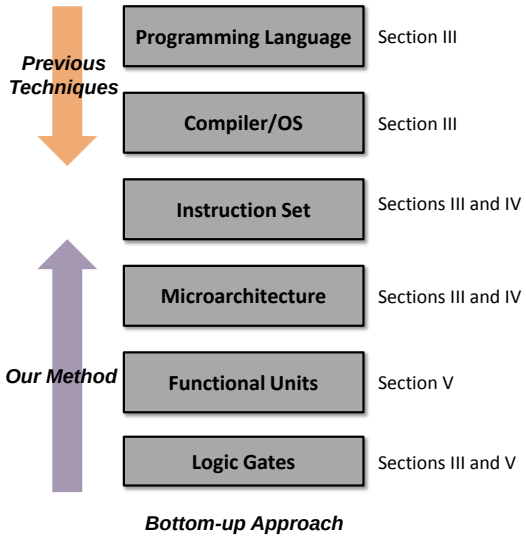

Fig. 1. Our bottom-up approach to security in high-assurance systems. Building a secure information flow secure hardware skeleton allows for strict control of all information flows all the way up the system hierarchy.

The remainder of this paper is organized as follows. In Section II we motivate the need for information flow security in mix-trusted reconfigurable systems. Section III presents the previous work in hardware information flow tracking techniques, specifically gate level information flow tracking (GLIFT). It also illustrates techniques applied to all layers in the system design stack as shown in Figure 1. Section IV discusses how a secure reconfigurable system can be designed based on findings in our previous work. It elaborates on a prototype system we developed using the mentioned bottom-up design methodology. Section V provides some fundamentals of gate level information flow tracking. This section focuses on how the Logic Gate level of the system stack can be developed with confidence. Section VI concludes the paper and presents some future research directions.

II. Reconfigurable Security and Motivation

Modern reconfigurable systems are typically designed with a number of mature building blocks known as *soft-cores* which generally come from vendors of varying trust. Some examples of these cores include AES encryption units, digital signal processors (DSP), memories, and Xilinx's MicroBlaze Processor [20]. The information flowing between these cores needs to occur only where the designer intends. For instance, trust needs to be upheld in a network interface core to ensure proper routing of information and data encryption cores must be proven to not leak the secret key through security holes including those from timing. From the designer's perspective, building a system with existing building blocks is tremendously easier than designing the system from scratch. As a result, these different cores should be placed into a system with confidence that they will not have any malicious effects on its operation.

Some potential threats that come up when using mix-trusted IP cores stem from the fact that their overall behavior is completely untrusted. Untrusted cores might be filled with malicious inclusions such as hardware trojans [4], [5] which can spawn unknown and harmful behavior that can cripple the integrity of other trusted portions of the system. Further, third-party cores can potentially learn secret information about classified components in the system if they all share a common bus. Providing strict information flow control in these reconfigurable systems is required if confidentiality and integrity of the systems is to be upheld.

Currently, it is difficult to ensure that an untrusted core is in fact behaving as expected. Some methods have focused on physically isolating cores and providing methods for routing over only statically defined channels. Our previous work [9] addresses this notion by using "moats" to isolate cores while allowing interconnect between cores only through pre-defined "drawbridges". In doing so, cores are expected not to leak secret keys (in the case of confidentiality) or be contaminated by untrusted data (in the case of integrity). However, even though cores are only understood to explicitly communicate through predefined channels, there is no guarantee about whether or not policy-violating information flows occur through implicit channels.

Previous work has shown that timing side channel attacks can be used to extract secret encryption keys from the latencies of caches [10] and branch predictors [13]. Cache timing attacks can obtain the secret key by observing the time for hit and miss penalties of the cache. Branch predictor timing channels are exploited in a similar manner where information is leaked through the latency of predicted and mis-predicted branches. Another exploit can be seen in a common bus where devices communicate implicitly through traffic (or lack of it) on the bus [14]. In order to have complete confidence that information is only flowing through the statically defined channels, strict information flow control needs to be guaranteed. The next section discusses some common techniques, specifically Gate Level Information Flow Tracking (GLIFT) which can be used to monitor the movement of all information in a system even those through timing channels.

III. Information Flow Tracking and GLIFT

A large amount of previous work has been done in the area of information flow security for complete systems. Numerous works have been done on information flow tracking specifically in hardware because monitoring information flows at this level allows for unintended flows to be identified without

significantly affecting system performance. This section discusses the previous research in information flow security for complete sytems and specifically focuses on GLIFT since it is the primary technique we used in previous and continuing analyses.

Information flow security focuses on monitoring the movement of data among different trust levels. Traditionally information flow security is guaranteed by ensuring that a particular information flow policy, such as integrity or confidentiality, is upheld. These policies can be modeled using a lattice $(L, \sqsubseteq)$ [19], where L is the set of security labels and $\sqsubseteq$ is a partial order between these security labels that specifies the permittable information flows. For example, consider the example lattices shown in Figure 2. Figure 2 (a) and (b) show two common binary lattices. For binary security lattices, the term *taint* is often used for the higher label on the lattice. For integrity, untrusted information is considered *tainted* in order to monitor if this taint violates a trusted (untainted) location. For confidentiality, taint is defined differently. The policy taints secret information to verify whether this leaks to an unclassified domain. Figure 2 (c) shows a confidentiality security lattice with multiple trust levels. Here confidentiality specifies that information may flow upward on this lattice but not downward. For simplicity, our analysis focuses on the binary lattices `trusted` $\sqsubset$ `untrusted` (a) for integrity and `unclassified` $\sqsubset$ `secret` (b) for confidentiality. For the two policies, this relation shows that information is allowed to flow from trusted to untrusted or from unclassified to secret respectively.

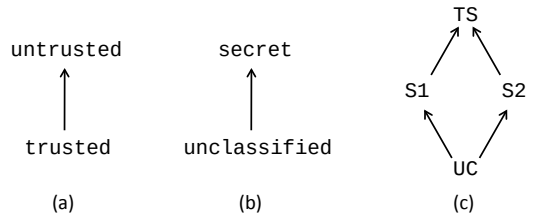

Fig. 2. Examples of different security lattices. (a) is a typical binary security lattice for integrity. (b) is a binary security lattice for confidentiality. (c) is a more complicated confidentiality lattice with multiple labels. Here TS is top-secret, S1 and S2 are two security levels which are less secure than TS but more private than UC (unclassified).

To be concrete, consider the code snippet shown in Figure 3. First, to understand explicit information flows and how they can be tracked, consider the assignment shown `y = z`. For integrity, using the lattice `trusted` $\sqsubset$ `untrusted`, this code is secure if $L(\mathsf{z}) \sqsubseteq L(\mathsf{y})$. In other words, this code is information flow secure only if the label assigned to `z` allows for information to flow into `y` without violating the integrity of `y`. Integrity in this situation is not violated as long as both `y` and `z` have the same label or `y` is untrusted and `z` is trusted. Confidentiality follows a similar procedure but using the lattice `unclassified` $\sqsubset$ `secret`. Using this lattice, the assignment is information flow secure if $L(\mathsf{z}) \sqsubseteq L(\mathsf{y})$. Meaning that information can only flow into `y` from `z` if `y` is at higher or equal security level to that of `z`. Implicit flows in this example follow a similar strategy except that they flow information indirectly to the variable. This particular code shows an implicit channel in the form of a branch. Here `x` leaks information to `y` because, depending on `x`, `y` will be assigned the value of `z`. For both confidentiality and integrity, implicit flows need to also be eliminated. In this particular example, to enforce integrity or confidentiality, the labels must adhere to $L(\mathsf{x}) \sqsubseteq L(\mathsf{y})$ in a similar manner as the explicit flow. Note that if integrity or confidentiality is to hold for the entire code, both the information flow constraints for the explicit *and* implicit flows must be enforced. Using this common model of information flow security, many implementations have been made to enforce this at all layers of the system design.

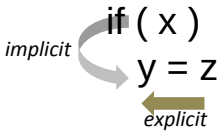

Fig. 3. Simple code snippet showing explicit information flow from `z` to `y` and implicit information flow from `x` to `y`.

The most common techniques for information flow security are implemented in programming languages using type based systems and in operating systems. Sabelfield and Myers [16] present a survey on the different programming language based techniques. Most work has been done in static compile based techniques which build off of the typing system of a language in order to enforce information flow security. These methods have shown to be effective and can even eliminate implicit channels due to conditional branches in execution. Jif [15] is a good example of such a type based system. Flume [17] has been shown to enforce information flow security using abstractions for operating system primitives such as processes, pipes, and the file system. Using theorem proving techniques, sel4 [18] has been shown to be fully verified for correctness. These schemes are often effective but are forced to abstract away the potential implicit flows that occur in hardware. Further, these schemes force the designer to comply with a new typing system (in programming language techniques) or reduce the overall system performance (in operating system abstractions).

To maintain system performance, information flow tracking has been proposed in hardware. The most common hardware information flow tracking strategies focus on the Instruction Set Architecture (ISA) and microarchitecture. One such technique called Dynamic information flow tracking (DIFT), proposed by Suh et al. [22], tags information from untrusted channels such as network interfaces and tracks it throughout a processor. They label certain inputs to the processor as "spurious" (tainted) and check whether or not this input causes a branch to potentially untrusted code. This technique has been shown to successfully prevent buffer-overflow [35] and format string attacks [36]. Raksha [23] is a DIFT style processor that allows the security policies to be reconfigured. Minos [24]

uses information flow tracking to dynamically monitor the propagation of integrity bits to ensure that potentially harmful branches in execution are prevented in a manner similar to [22].

These previous techniques are effective at ensuring that potentially harmful branches in control flow are prevented or guaranteeing the integrity of critical memory regions. However, these methods target a higher level of abstraction (from the microarchitecture up as shown in Figure 1) and cannot be used to monitor the information flows in general digital hardware. For this reason, these methods also fail to detect hardware specific side channels in the form of timing. GLIFT provides a solution for tracking information flows, including those through timing channels, in general digital hardware. GLIFT works by tracking each individual bit in a system as they propagate through Boolean gates. This is done using an additional tag bit commonly referred to as *taint* and tracking logic which specifies how taint propagates. Information is said to flow through a logic gate if particular *tainted* inputs have a chance to *affect* the output.

Taint is a label associated with each data bit in the system which indicates whether or not this particular data bit should be tracked. If integrity is a concern, untrusted information is tainted to ensure that this tainted information does not flow to a trusted location. In the case of confidentiality, secret information is tainted to monitor whether it leaks to a public domain. Taint is propagated whenever a particular tainted data bit can affect the output. In other words, if the output of a function is dependent on changes to tainted inputs, then the output is marked as tainted.

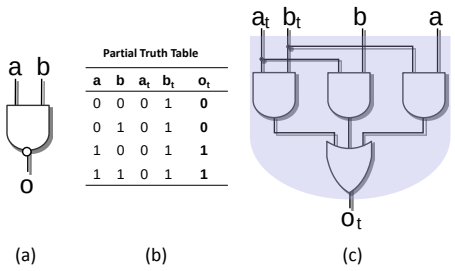

Fig. 4. (a) A two-input NAND gate. (b) Truth table of two-input NAND gate with taint information (not all the combinations are shown). (c) The corresponding tracking logic of two-input NAND gate is $ab_t + ba_t + a_tb_t$. Every change at the input of the gate is precisely tracked at the output.

For example, consider a simple 2-input NAND gate as seen in Figure 4 (a) and its corresponding tracking logic as shown in Figure 4 (c). For a NAND gate, only particular input changes will result in a change at the output. Specifically, consider the case in which $a = 0$ and $b = 1$. Here changing the value of b will cause no change at o since $a = 0$, meaning that there is no information flowing from b to o. If b were to be tainted ($b_t = 1$) and a untainted ($a_t = 0$) in this case, o would be untainted ($o_t = 0$) since the tainted input does not affect the output. A subset of all such combinations can be seen in Figure 4 (b). Using the full truth table, a function can be derived for all similar input combinations into a tracking logic function as shown in Figure 4 (c). Since NAND is functionally complete, the tracking logic for any digital circuit can be derived by constructively generating the tracking logic for each gate. In other words, given a circuit represented as NAND gates, the circuit can have complete information flow tracking by interconnecting the tracking logic for each individual NAND gate. This results in a design that precisely tracks the information flow of each individual bit. As mentioned, GLIFT is a useful tool for analyzing any digital hardware because it exposes all information flows explicitly. The next section will discuss how we used GLIFT and related techniques to develop a prototype system that was verified to be information flow secure.

IV. Information Flow Secure System Design

As reconfigurable systems become more complex, it is very common to have multiple mix-trusted IP cores existing in the same design. Previous work makes efforts to ensure physical isolation between different IP cores and have interconnect only through known channels [9]. However, since information can often flow through difficult to detect timing channels, deeper analysis is required in order to guarantee complete information flow isolation between mix-trusted IP cores in these systems. This section discusses how a secure reconfigurable system can be designed from the bottom-up (Figure 1) using GLIFT and related techniques.

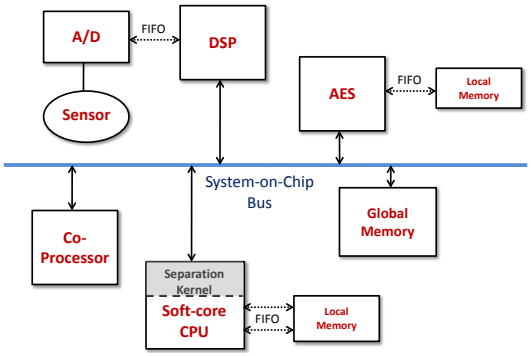

Fig. 5. A common reconfigurable system which consists of a soft-core processor, a global system-on-chip bus, and various peripherals. Interaction between each of these components must be regulated to ensure that information is flowing only to where the designer intends it.

Typical embedded reconfigurable systems consist of a soft-processor core running a microkernel or more robust operating system. Interacting with this processor is typically a wide range of peripherals from digital signal processors (DSP), AES encryption cores, memories, and even other processor cores. A common reconfigurable based system can be found in Figure 5. Here a soft-core processor exists with several different peripherals all interacting over a common bus. There are also components interacting with their own memories (AES core) or with external sensors (DSP core). Common bus protocols in such systems include ARM's AMBA, Inter-integrated circuit

protocol (I^2C), and the universal serial bus (USB). The soft-core processor is typically the "heart" of the system and manages most of the interaction and runs application software. Typical embedded systems consist of a small microkernel which manages the execution of mix-trusted applications. As a result, it is essential to provide information flow guarantees in the processor core, its interaction with various peripherals over commodity buses, and in the microkernel itself since it arbitrates between mix-trusted executions. The following subsections introduce these different components and discuss our general testing flow.

A. GLIFT Test Method

The typical GLIFT based testing method can be seen in Figure 6. Here a design is modeled in a hardware description language (HDL) such as Verilog or VHDL. This testing flow is general enough to target any model in Verilog or VHDL, however finite state machine (FSM) representations are much easier to analyze for implicit timing channels since state transitions commonly occur every cycle.

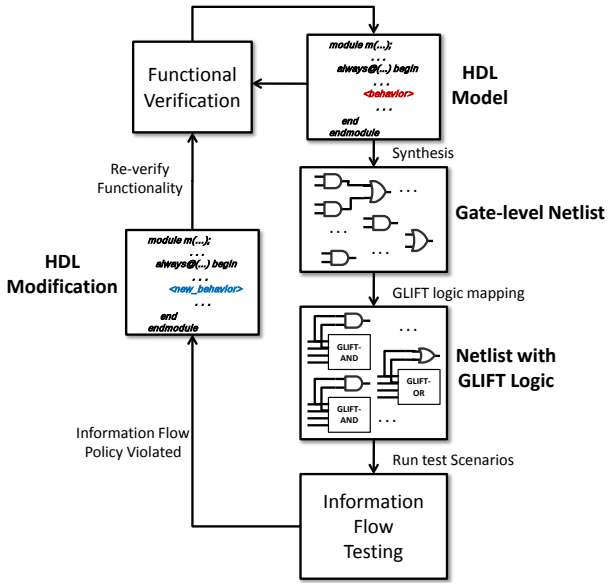

Fig. 6. Information flow test method for general digital hardware. Designs are represented as a finite state machine in Verilog or VHDL and processed through the testing flow.

As Figure 6 shows, the design enters the analysis flow as Verilog or VHDL. At this stage, the hardware model undergoes typical functional verification to ensure that the design conforms to the correctness of its specification. Note that at this stage, information flow guarantees are not yet being tested.

After synthesis, the design is equipped with GLIFT logic, which allows all of its information flows to be analyzed. There are many methods for generating this tracking logic for a given circuit, but this particular analysis uses a constructive approach. In other words, every gate in the system has logic attached to it individually in a linear fashion. This design equipped with GLIFT logic is now capable of being analyzed for unintended information flows. At this stage, test vectors are run on the hardware to see if it violates the information flow policy. If the policy is violated, the description of the hardware is modified and once again undergoes functional verification and proceeds through the testing flow again.

Making modifications to the hardware model is not obvious and generally requires a form of time-multiplexing to prove the absence of timing channels from the design. This has shown to be effective when analyzing processor cores and bus controllers as subsequent sections discuss in more detail.

B. Processor Core

The processor core is required to handle majority of the application execution in the system. Since the processor core acts as the central unit of the system, it requires very strong information flow guarantees. It needs to be able to support mix-trusted program execution and ensure that its state is recovered or purged such that it does not leak any information.

An execution lease architecture, as proposed by Tiwari et al. [30], provides an effective method for guaranteeing the integrity and confidentiality of the system following mix-trusted context switches. As shown in Figure 7, this architecture works by *leasing* out the hardware for an untrusted application to execute for a fixed amount of time and with restricted memory bounds. In other words, the processor restricts lower trust programs to a space-time sandbox. This allows the leasee to execute any code it desires during this fixed time slot. Once the time slot expires, the leaser restores the program counter back to a known value and the system state is restored.

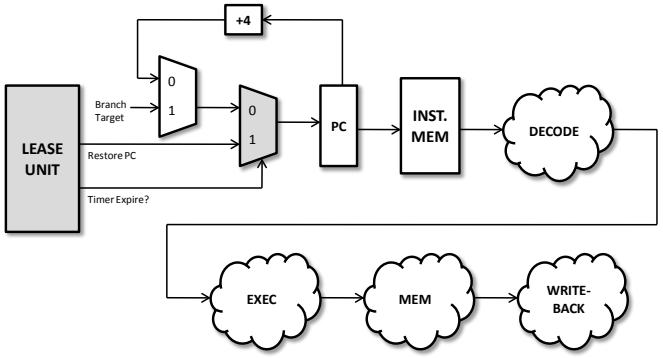

Fig. 7. The execution lease architecture fits directly into the typical 5-stage pipeline. Here the lease-unit restores the program counter upon the expiration of a lease. During a lease, the processor operates as normal. Restoring the system after a lease allows the architecture to execute untrusted code without leaking information.

Additionally, the execution lease architecture uses a stack of nested leases to allow for a larger granularity of trust. This stack of leases is managed in the execution lease unit. This unit keeps track of the current memory and time bounds. If the currently executing program wishes to execute untrusted code, it stores its current program counter in the lease unit, sets the memory and time bounds in the lease unit, then jumps to the untrusted code. Since there is a stack of nested leases, multiple

programs can lease out the architecture to other less trusted ones. It should be noted that untrusted code cannot lease out to trusted code since the untrusted application can potentially observe the state once they lease expires, thus violating the information flow policies.

Specifically, this architecture implements these leases using two instructions known as `set_timer` and `set_membounds` which set the amount of time on the lease and the memory bounds respectively. The architecture first begins in a trusted state and this trusted state can be leased out to execution contexts which are at a lower trust level (in the case of integrity). Using the stack of leases, each subsequent lease can be further leased out to lower trust levels using the mentioned instructions.

The execution lease processor provides information flow guarantees by dynamically tracking information flows using GLIFT but lacks much of the performance optimizations of a modern processor including caches and pipelining. Our Star-CPU [31] builds off of the execution lease processor with a few additional features including caches and pipelining. Caches, pipelines, branch predictors and other stateful elements in a processor open the doors for potential timing attacks. Such attacks on caches have been shown to leak secret keys [10], [11] through the latency of caching operations. The attack works by an untrusted process first filling the contents of the cache. Subsequently a trusted process runs and upon a context switch back to the untrusted process, the untrusted process is able to observe which of its cache lines were evicted and extract the secret key as a result. The issue results from the mix-trust sharing of data in the cache and some current methods have made attempts at solving this problem [11] but some have found these new designs to still be partially vulnerable [12]. Our Star-CPU solves this issue by requiring that contents of the cache be strictly controlled by a trusted entity (separation kernel) and that strict partition in the cache is enforced. By having a trusted kernel controlling the contents of the cache, clearing and strict partitioning is enforced between context switches. This ensures that mix-trusted processes never learn information about one another through caching behavior.

Our Star-CPU provides further optimization over the execution lease CPU by providing pipelining. In a secure system, pipelines can leak information through timing channels due to non-deterministic behavior caused by branches and memory stalls. To ensure that such information does not leak between two processes of different trust, our CPU ensures that the pipeline is appropriately flushed between context switches. This ensures that such dynamic behavior does not cause information flows between programs of varying trust through implicit timing channels.

With any processor, it is desirable to communicate with various peripherals to gather data from sensors, read memories, or interact with DSPs. These peripherals all come from different trust levels and ensuring that data flows only to where the designer intends is required. The next subsection discusses these details further by analyzing the information flow characteristics of I^2C and USB.

C. Secure Input-Output for System Interaction

As mentioned, it is very common for reconfigurable systems to offer the flexibility for system designer to use many IP cores from many different vendors all with varying trust. Aside from the central processor core, many of these mix-trusted cores integrate directly into the system bus architecture and are frequently swapped out for either updated or completely different cores. For high-assurance systems, it is required that the communication between these various cores is information flow compliant. This section discusses our test methodology when analyzing information flow properties of bus protocols using GLIFT and briefly discusses our analysis of two common protocols: I^2C and USB.

1) Information Flows in I^2C: The inter-integrated circuit (I^2C) protocol was first developed by Philips. The protocol consists of a simple two wire bus with a serial data line (SDL) and serial clock line (SCL). In I^2C, devices all sit on a global bus with the master so explicit information clearly flows between all devices, since they can explicitly snoop the communication on the bus. However, as some of our previous work has shown [28], information can leak through difficult to detect timing channels.

To observe these timing channels, we processed our I^2C controller FSM through the testing flow shown in Figure 6. Our test consisted of a scenario in which master performs a write transaction with an *untrusted* slave and subsequently a write transaction with an *trusted* slave. This particular testing scenario is concerned with non-interference [6] (data integrity), but the analysis certainly works for confidentiality. As mentioned, I^2C operates on a global bus, so information clearly flows explicitly between devices on the bus since they are capable of openly snooping. However, even if devices are assumed to not snoop the bus, the state at which the master is left in leaks information from the untrusted slave to the trusted slave. This implicit information leak is in the form of a timing channel.

Following the communication with the untrusted slave, the master changes state and ultimately ends up in an untrusted state due to its interaction with an untrusted device. Subsequent communication with a trusted devices causes this untrusted information to flow into the trusted device. Since we are concerned about guaranteeing non-interference, we need to ensure that no information flows from the untrusted device to the trusted one through any channel. To eliminate all channels, we need to first bound the amount of time in which devices can communicate with the master. Secondly, we need to enforce that the master is returned back to a "known" state prior to communication with other devices. This solution can be seen in Figure 8 where an adapter is placed in between each device and the global bus. This adapter allows for a device to be connected to the bus only in a specific time slot. Once this time slot expires, the execution lease unit shown restores the master back to a known state to eliminate any possibility of an implicit flow. It should be noted that such an adapter can serve two purposes: 1) for information flow control and 2) for fault

tolerance via a fault control unit (FCU) as discussed by [33]. Such dual usage justifies the hardware overhead and extends the confidence in the bus system to not only be fault tolerant but also information flow secure.

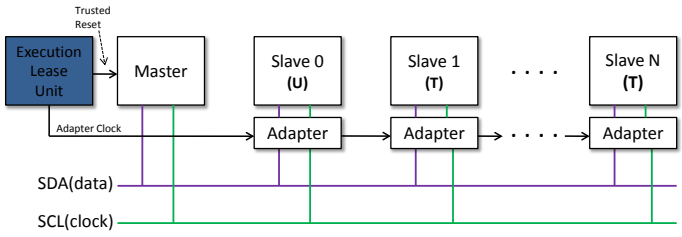

Fig. 8. I^2C bus system with an adapter which restricts information flows between devices. (U) represents an untrusted device and (T) a trusted one. After a time slot expires, the master is restored back to a known state prior to communicating with other devices.

2) Information Flows in USB: We chose to also analyze USB because of its different bus topology. USB operates as a star-tiered topology and does not operate on a global bus like I^2C. Since the bus is not global, there are no explicit information flows between devices because the architecture does not allow them to communicate with one another. However, as we will discuss, similar implicit information flows leak information between devices as seen in I^2C.

To identify the timing flows in USB, we modeled a USB system in Verilog and processed it using the flow shown in Figure 6. Our testing scenario consists of a USB host and two USB devices; one untrusted and the other trusted. The host performs a write transaction to an untrusted device and then subsequently a write to the trusted device. As in I^2C, this particular scenario focuses on non-interference to monitor the integrity of the trusted device. Since the devices on a USB bus have no way to interact with one another, there are no explicit flows. However, in a similar manner as I^2C, information implicitly flows between devices in the form of timing.

In this scenario, since the host first communicates with an untrusted device, the state in which it is left in unavoidably becomes untrusted. Subsequent communication with a trusted device causes this untrusted information to flow into the trusted device. Although this timing channel occurs in a nearly identical way to that of I^2C, to guarantee the absence of information flows requires a slightly different approach since USB is a much different bus architecture. In this case, the host operates using two different states, one which is untrusted and the other trusted. If the master is doing trusted communication it uses the trusted state and the untrusted state for untrusted communication. Swapping between the states is done in a similar manner as a context switch where the state is replaced yet none of the hardware is required to be replicated.

Having strict information flow isolation on a bus is only as good as the software which manages it. The next subsection discusses how a separation kernel can be used to allow for secure software executions and have secure management of both the processor and I/O.

D. Separation Kernel

Separation kernels are essential when managing many multi-process environments of varying trust. Separation kernels were first introduced by J. Rushby [29] as a way to show provable isolation between "partitions" and allowing communication through only predefined channels. Such a model is required when managing mix-trusted applications in a highly reconfigurable system. Reconfigurable systems will often continuously evolve their application source code, so it is essential that a managing mechanism can ensure information flow security between these applications.

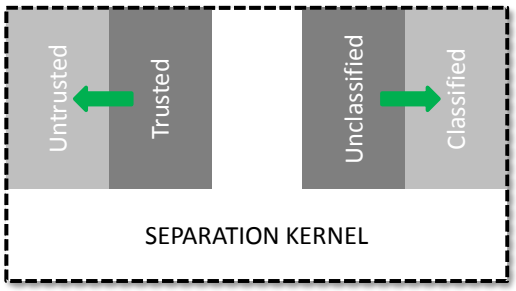

Fig. 9. A common separation kernel structure. A separation kernel manages the execution of different partitions of varying trust. Partitions are allowed to communicate over only predefined channels. For example, a trusted partition can write to an untrusted one and a classified partition can read from an unclassified one.

A high level overview of the organization of a separation kernel can be seen in Figure 9. As shown, there are different partitions which have varying trust. The separation kernel by definition should allow explicit communication only through defined communication channels. For example, communication between two processes should only exist through an IPC mechanism defined a priori. The separation kernel is required to manage all *leasing* parameters as enforced by the previously mentioned information flow secure CPU in Section IV-B.

Our implementation of such a microkernel in our information flow secure prototype system [31] requires that information flow security is preserved between context switches. Our separation kernel operates at the highest level of trust and is in charge of initially leasing out the architecture to less trusted execution contexts using the `set_timer` and `set_membounds` instructions. Since the kernel is at the highest level of trust, it is essential that while managing context switches it does not leak information between partitions. To enforce this, our separation kernel must not only have complete control over the microarchitectural state, but must also avoid pipeline stalls and cache misses since such non-determinism can leak information between partitions of varying trust upon a context switch.

Upon a context switch from an expired lease, the separation kernel follows a deterministic procedure. After a delay to flush the pipeline of the CPU, it commits the `set_partitionID` instruction for itself which allows it to access the memory of all partitions to store their state. This instruction activates the

portions of the cache that are reserved for the currently executing partition. Once activated, the kernel obtains the current PC of the expired partition and stores it in the memory location reserved for this specific partition. Once the previous partitions state is saved, the kernel sets up the new context using the `set_partitionID`, `set_timer`, and `set_membounds` which set up the cache bound, lease duration, and new partition memory bounds respectively. Once the context for this new partition is set, the kernel loads the partitions PC and jumps to it.

Since the separation kernel operates in its own reserved cache partition, it never misses the cache. It also contains no branches that introduce non-deterministic pipeline delays. Such deterministic operation allows the kernel to perform a context switch between any two partitions without leaking any information about the preempted partition. Further, the behavior of the kernel must be strictly independent from any partition since it operates at the highest level of trust. To understand the importance of having deterministic context switches, suppose that the kernel did not operate in its own cache partition. If this was the case, the time in which a context switch occurred is directly dependent on what occurred in the old partition. Not only does information leak from the old partition to the new partition through this timing channel, but information also leaks from the less trusted partition into the kernel itself since it indirectly influenced its behavior. Since we are interested in adhering to information flow security through *all* channels, such timing channels need to be eliminated.

The kernel operates at the highest level of trust in the system and builds directly off of our secure hardware skeleton, thus its information flow security is also statically verified. All levels of our system are verified to be information flow secure using our $*$-logic technique, which is discussed in the next section.

E. Static Verification of Information Flow Security in Hardware

To statically verify that our complete system adheres to our information flow policy. We also developed a tool, known as $*$-logic (star-logic) [31], which is used to statically verify our policy.

This tool works by first building an abstract representation of the original design which gives the flexibility to specify any part of the system as unknown or $*$. This is required because in most reconfigurable systems the specific applications or operations of the system are not necessary known a priori. Once in this abstracted state, information flow tracking labels (such as trusted and untrusted) are assigned to the individual bits of the system. This abstracted system equipped with information flow tracking logic is simulated for all possible combinations. Since the only known part of the system at this stage is the separation kernel itself, all states can be exhaustively enumerated to ensure complete information flow security. As a result of this, this abstraction allows for the processor and system to be statically verified even if much of it is left unspecified at design time. Using our tool, we showed that this complete system is verifiably information flow secure given that the software base (separation kernel) was written by a trusted source.

Another static technique that works well for this verification is our tool Caisson [32]. Caisson is an HDL that uses static type checking to ensure information flow security. This static type checking is very similar to the previous IFT techniques in programming languages [16] as discussed in Section III. The key difference is that this language is intended for designing hardware, so the static type checking translates to secure hardware upon compilation. In other words, once hardware is designed in Caisson and passes the security type checker, our compiler generates information flow secure and synthesizable Verilog HDL. In our previous work [32], we have shown that a complete information flow secure processor can be designed using Caisson's static guarantees.

In both cases, this static verification requires some assumptions. First, the designer needs understand what resulted in an information flow or caused an error in the type checker and have the ability to make modifications to the system so that it is information flow compliant. When analyzing flows in hardware, using GLIFT or $*$-logic, designers should also be confident that the observed information flows are in fact not false positives. The next section discusses these issues to give the hardware designers better confidence in the information flow characteristics of the hardware.

F. Future Research in Information Flow Secure Systems

There is much room for improvement for the development and testing of information flow secure systems at every level of the design phase. In these systems, it is often desirable to re-use existing processor cores such as MicroBlaze [20] or Nios II [21] rather than designing one for security. In fact, most reconfigurable systems use these soft-core processors for applications which do not require a large software base and are mostly dataflow intensive. In this case, information flow control through the processor and between peripherals should be guaranteed. Future research directions should work towards monitoring the movement of information both intra- and inter-IP cores even if the RTL is not released. A question we are working to answer is: How do we leverage both software (for monitoring the movement of information through the embedded code) and hardware (for monitoring the movement of information between peripherals) techniques to build secure systems with exisisting procesors?

More research should also be taken in the security of the bus system. The bus system is the central location for all communication between the various cores involved in the system. Since many reconfigurable systems frequently interconect different components in the bus architecture generally from varying trust levels, having an information flow secure bus architecture is required if the overall system is concluded to be secure. Our previous work has shown a couple of promising methods for designing a secure bus architecture with timing enforcement in adapters for I^2C and root hubs in the host for USB. However, more research needs to be taken into using existing system-on-chip protocols with only

minimal modification. Future research should be tagetted at the complicated and robust bus architectures used in modern systems, such as ARM's AMBA. Some common questions are: What sort of performance trade off do our methods have? How hard is it to prove the information flow properties for complex bus systems? What knowledge do we need to design provably information flow secure protocols? It should also be mentioned that our previous analyses of I^2C and USB involved specific testing scenarios. Meaning, we were only able to show information flow security for specific bus transactions. Future research directions are targeted at using our most recent technique, $*$-logic as discussed in Section IV-E, to statically verify the absence of information flow for any bus transaction.

The separation kernel also has future research for information flow security. In a reconfigurable environment it would be desirable to have the flexibility to modify or configure the properties associated with partitions or even the number of partitions in the kernel. Currently, making these changes requires that the system be completely re-verified for information flow security. Future research should be put into providing abstractions such that more reconfigurability is allowed in the system kernel to avoid re-verification. Is it possible to allow dynamic reconfigurability of portions of the kernel? How restrictive do we need to be without violating its integrity?

V. GLIFT Generation in Reconfigurable Systems

Up to this point, we have discussed how an information flow secure reconfigurable system can be designed with the help of GLIFT. However, much of the details of GLIFT and the details of its functionality have been left out. This section first discusses two GLIFT logic generation methods, namely the *brute force* and *constructive* methods. It also focuses on a preciseness problem that arises when using the constructive method.

A. The Brute Force Method

The brute force method for generating GLIFT logic follows straight from the definition of information flows and, as the name suggests, is a "brute force" approach with its complexity being exponential. This method works by changing an input to a function and observing whether or not this caused a changed at the output. If a change occurred, information is said to flow from the input to the output by definition of an information flow. For each such case we add a logic term to the GLIFT logic function. Once all input combinations are checked, we will have generated complete GLIFT logic which precisely tracks information flows through the original function.

The complexity of this algorithm is $O(2^{2n})$ where n is the number of inputs in the original function under test. The interested reader should refer to some of our previous work [34] to find more details of this proof and analysis. Although the brute force method does in fact generate a precise GLIFT function, (i.e. it indicates the presence of an information flow *iff* one actually occurred), its computational complexity makes it impractical for any reasonably large designs. Another method, known as the constructive method, operates in linear time but with a slight trade-off in precision.

B. The Constructive Method

The constructive method is a much less complex approach than the brute force method. It works by generating GLIFT logic for each primitive in the system compositionally. This is done by creating a library for each primitive in the design and mapping this primitive to its corresponding GLIFT logic in a similar manner as technology mapping. For example, GLIFT logic primitives for *AND*, *OR*, and *NOT* can be built into a library. As the circuit is being processed, each gate has its corresponding GLIFT logic replaced using the GLIFT primitives from the aforementioned library.

Figure 10 shows the constructive method when used on a 2-input Multiplexer. First, the logic equation represented as a network of *NOT*, *AND* and *OR* gates. Then, these logic primitives are replaced using the GLIFT primitives in the library as previously mentioned. Finally, proper connections are made to complete the shadow logic circuit.

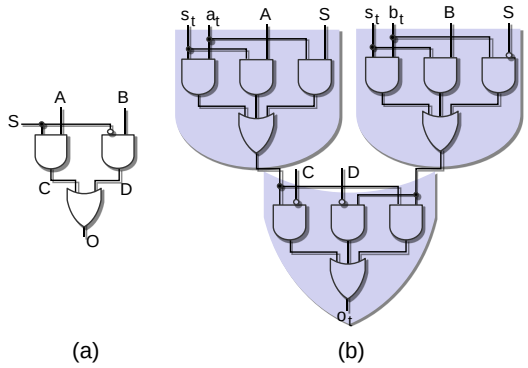

Fig. 10. (a) A 2-to-1 multiplexer. (b) GLIFT logic of 2-to-1 multiplexer generated using the constructive method.

However, generating GLIFT logic using the constructive method is not always guaranteed to be precise. Meaning, a GLIFT logic function generated with this method will often indicate that a flow of tainted information propagated from the input to the output when it in fact no such flow occurred. In other words, the GLIFT logic will have false positives indicating the false presence of information flows.

For example, Table I shows the number of "1's" (minterms) in the GLIFT logic truth tables for a 4-bit adder generated by the two discussed methods. We can see that the number of minterms for the brute force method is less than or equal to that of the constructive method. This means that the constructive method more frequently indicates that information flowed from the input to the output of the logic function. Since the brute force method generates precise GLIFT logic by its definition, the constructive method is actually overly conservative because it contains false positives.

Such additional minterms are false positives that indicate that a flow of information has occurred when in fact it has not. Taint can quickly propagate throughout the system, e.g.,

TABLE I
MINTERM COUNTS OF SHADOW LOGIC FUNCTIONS OF A 4-BIT ADDER GENERATED BY THE BRUTE FORCE AND CONSTRUCTIVE METHODS.

Method	sum[0]	sum[1]	sum[2]	sum[3]	cout
Brute Force	229376	241664	246272	248000	208160
Constructive	229376	245760	251648	250656	227864

a tainted state machine can taint the whole design in just a few clock cycles or a tainted PC (Program Counter) will quickly cause every bit of information in the processor to become tainted. When a conservative shadow logic function is used for taint propagation, the entire system can get into a tainted state when in fact it is not tainted. At this point, a declassification such as what is presented in [30], from a separation kernel with the highest security level is required to recover the system to a usable state. Generally speaking, being conservative is safe but frequent declassification will make a system unusable since an overbearing number of false-positives will continuously indicate that the system is untrusted (non-interference) or leaking confidential information (confidentiality). Section V-C discusses the imprecision problem and overviews solutions to it.

C. Imprecision Problem

As mentioned, the constructive method tends to report false positives in information flows for certain functions. The precision problem is important to understand especially when information flows are to be understood with high confidence. In a reconfigurable system, if the information flow tracking logic used frequently indicates that the system is in an untrusted state, then proving safe interaction between mix-trusted subsystems will essentially become impossible. Understanding how to reduce the amount of false positives in the analysis logic is essential to building an information flow secure system.

The constructive method in general produces more false positives than the brute force method making it overly conservative. A good example of these false positives can be seen in a 2-input multiplexer if GLIFT logic for this circuit is generated using the constructive method as shown in Figure 10.

The overly conservative result occurs when the select line of the multiplexer is tainted. In this case, the GLIFT logic indicates that the result is tainted regardless of what the inputs are. This is certainly the case if the inputs to the multiplexer are different since changes at the tainted select line will cause a change at the output of the function. However, if both inputs are the same, then the select line does not actually affect the output since the output will remain the same regardless of what the select line's value is.

To build a better intuition, consider the Karnugh Map in Figure 11 when S is tainted with A and B both untainted and logical 1. If the GLIFT logic is generated constructively for this circuit the two terms shown in black boxes will have GLIFT logic generated in addition to the OR gate that takes the disjunction of the two terms. Independently, the GLIFT logic for each of these terms is required to assume that the output of the function changed whenever the value of one of these terms changes in order to ensure "correctness". However, it can be seen that if the select line (S) changes and the values of A and B are both 1, the output value does not change (i.e. f remains 1), so there is in fact no tainted information flowing from the select line to the output.

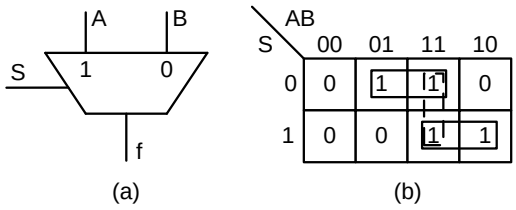

Fig. 11. Karnaugh map of a 2-input multiplexer. The initial function $f = SA + \overline{S}B$, when shadowed constructively, is not precise. The dotted box indicates the additional term AB that must be added to the original logic function to insure its constructively derived shadow function is precise.

The root of this problem stems back to logic hazards in digital circuits and requires that the circuit have all *prime implicants* before having its GLIFT logic generated using the constructive method. The extra prime implicant in this example is indicated by the checkered box in the Karnaugh Map. The interested reader should refer to our previous work on this topic [34] for a formal proof of how this imprecision problem can be solved and how it requires a trade off between area and delay. The key detail from this precision problem is that some circuits (e.g. a 2-input multiplexer) require a careful information flow analysis in order to create a system which will be information flow secure with the best flexiblity. Since multiplexers are essential building blocks for any digital system, poor management of this precision problem easily results in an explosion of false positives which makes creates tremendous burdens during information flow verification when trying to understand the cause of the information flows.

D. Future Research in GLIFT Generation

Continuing research in GLIFT logic generation should focus on methods for better optimization for area, delay, and simulation time. It should also concentrate on methods for better understanding the precision problem. Optimizing for area, delay, and simulation time can be done using different encodings techniques for the GLIFT logic. This will produce state reductions resulting in smaller and more efficient tracking logic. How can we reduce the overhead of the tracking logic so that it is suitable to be deployed dynamically? Can we overload the logic such that it serves more than one purpose?

Another research area should focus on the precision problem in more detail. Being completely conserative is ineffective since the solution ends up being similar to physical isolation. Full precision is likely excessive since it results in tremendously large overheads in area and delay. Specifically, how much precision does a given application need to adequately track its information flows? Can this amount be determined and possibly quantified for a given system? Answers to these

questions will allow more effecient use of GLIFT logic and will help designers better understand the security holes in their designs.

VI. Conclusions and Future Research

As the number of reconfigurable systems increases, the designs they implement are becoming even more complex. As these systems continue to benefit from COTS IP cores from vendors of varying trust bases, the need to ensure safe interaction between different components is critical. Information flow control is essential in these systems to guarantee the integrity of trusted components and confidentiality of secret data. GLIFT is a useful technique for monitoring information flows even those through implicit timing channels. This bottom-up approach to secure reconfigurable system design allows for strong information flow guarantees through all channels including timing. We have shown how this bottom-up approach can be used to build a verifiably information flow secure system with the help of gate level information flow tracking and related techniques.

Future research should focus on allowing more flexibility in the reconfigurability of the system. We have shown that current methods allow for a good starting point, where a secure hardware foundation can be multiplexed with mix-trusted cores and software and still be information flow secure. Strong information flow guarantees should be allowed for reconfigurable systems which use existing IP cores. New research needs to leverage both IFT in software and hardware to monitor the information inter- and intra-processor to allow the reuse of existing processor cores. In addition, our current methods have shown that mix-trusted cores can exist on the same bus without violating information flow security. However, more intricate bus protocols and architectures should be tested to show that this scales to modern systems. Lastly, kernels in such systems should have the flexibility to be reconfigured without complete system re-verification. Further abstractions are necessary to allow enough flexibility in the kernel design to accommodate more configurable and robust applications.

References

[1] *Common criteria for information technology security evaluation*. http://www.commoncriteriaportal.org/cc/

[2] *What does cc eal6+ mean?* http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean/

[3] *The integrity real-time operating system*. http://www.ghs.com/products/rtos/integrity.html

[4] Y. Jin and Y. Makris. *Hardware Trojan detection using path delay fingerprint*. IEEE International Workshop on Hardware-Oriented Security and Trust, Anaheim CA, 2008.

[5] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey. *Hardware Trojan horse detection using gate-level characterization*. Proceedings of the Design Automation Conference, 2009. vol., no., pp.688-693, 26-31 July 2009

[6] J. A. Goguen, J. Meseguer, *Security Policies and Security Models*. pp.11, IEEE Symposium on Security and Privacy, 1982

[7] D. Bell and L. LaPadula. *Secure computer systems: Mathematical foundations*. Technical report, Technical Report MTR-2547, 1973

[8] D. Federal Aviation Administration (FAA). *Boeing model 787-8 airplane; systems and data networks securityisolation or protection from unauthorized passenger domain systems access*. http://cryptome.info/faa010208.htm

[9] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine. *Moats and Drawbridges: An Isolation Primitive for Recongurable Hardware Based Systems*. In Proceedings of the Symposium on Research in Security and Privacy, Oakland, May 2007

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

[11] Z. Wang and R. Lee. *New cache designs for thwarting cache-based side channel attacks*. In Proceedings of the 34th International Symposium on Computer Architecture, San Diego, CA, June 2007

[12] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. *Deconstructing new cache designs for thwarting software cache-based side channel attacks*. In Proceedings of the 2nd ACM workshop on Computer security architectures, CSAW 08, pages 2534, New York, NY, USA, 2008. ACM

[13] O. Accigmez, J. pierre Seifert, and C. K. Koc. *Predicting Secret Keys via Branch Prediction*. In Cryptology, The Cryptographers Track at RSA, pages 225-242. Springer-Verlag, 2007.

[14] W. M. Hu. *Reducing Timing Channels by Fuzzy Time*. In Proceedings of the Symposium on Research in Security and Privacy, Oakland, May 1991.

[15] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. *Jif: Java information flow*. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[16] A. Sabelfeld and A. C. Myers. *Language-based information-ow security*. IEEE Journal on Selected Areas in Communications, 21:2003, 2003

[17] M. Krohn and E. Tromer. *Noninterference for a practical difc-based operating system*. In Proceedings of the 2009 IEEE Symposium on Security and Privacy, 200

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verication of an os kernel. In SOSP 09: 22nd Symposium on Operating Systems Principles, pages 207220, NY, USA, 200

[19] D.E. Denning. *A lattice model of secure information flow*. Comm. ACM 19, 5 (May 1976), 236-243.

[20] *Xilinx Microblaze Soft-core Processor*. Available Online: http://www.xilinx.com/tools/microblaze.htm

[21] *Nios II Embedded Processor*. Available Online: http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[22] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. *Secure Program Execution via Dynamic Information Flow Tracking*. In ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, pages 85-96, New York, NY, USA, 2004. ACM Press.

[23] M. Dalton, H. Kannan, and C. Kozyrakis. *Raksha: A Flexible Information Flow Architecture for Software Security*. In 34th Intl. Symposium on Computer Architecture (ISCA), June 2007.

[24] J. R. Crandall and F. T. Chong. *Minos: Control Data Attack Prevention Orthogonal to Memory Model*. In Proceedings of the International

Symposium on Microarchitecture (MICRO), 2004

[25] 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 Transactions on Software Engineering, 17(11):11471165, 1991.

[26] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. 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 (ASPLOS), 2009.

[27] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood and R. Kastner, *Theoretical Analysis of Gate Level Information Flow Tracking*, In proceedings of the 47th Design Automation Conference (DAC'10), June 2010.

[28] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, *Provable Information Flow Isolation in I^2C and USB*, In proceedings of the 48th Design Automation Conference (DAC'11), June 2011.

[29] J. Rushby, *Proof of Separability*, In Proceedings of the 5th International Symposium on Programming, Springer Verlag LNCS Vol. 137, pp. 352–367, Turin, Italy, 1982

[30] 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*, Proceedings of the International Symposium on Microarchitecture (Micro), December 2009. New York, NY

[31] M. Tiwari, J. Oberg, X. Li, J. K. Valamehr, T. 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*, Proceedings of the International Symposium of Computer Architecture. (ISCA) June 2011. San Jose, California

[32] X. Li, M. Tiwari, J. Oberg, F. T. Chong, T. Sherwood, and B. Hardekopf. *Caisson: A Hardware Description Language for Secure Information Flow*. In Proceedings of Programming Language Design and Implementation (PLDI 2011)

[33] J. Rushby, *Bus Architectures For Safety-Critical Embedded Systems*, Proceedings of the First Workshop on Embedded Software (EMSOFT), 2001, Lake Tahoe, CA.

[34] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. *Theoretical Fundamentals of Gate Level Information Flow Tracking*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).

[35] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, *StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks*. Proceedings of the 7th conference on USENIX Security Symposium, p.5-5, January 26-29, 1998, San Antonio, Texas

[36] T. Newsham. *Format String Attacks*. Guardent, Inc. September 2000. http://hackerproof.org/technotes/format/formatstring.pdf