Information Flow Isolation in I2C and USB

Jason Oberg
Computer Science and Engineering, UC San Diego
jkoberg@cs.ucsd.edu

Wei Hu*
Automation, Northwestern Polytechnical University, Xi'an China
w3hu@cs.ucsd.edu

Ali Irturk
Computer Science and Engineering, UC San Diego
airturk@cs.ucsd.edu

Mohit Tiwari
Computer Science, UC Santa Barbara
tiwari@cs.ucsb.edu

Timothy Sherwood
Computer Science, UC Santa Barbara
sherwood@cs.ucsb.edu

Ryan Kastner
Computer Science and Engineering, UC San Diego
kastner@cs.ucsd.edu

ABSTRACT

Flight control, banking, medical, and other high assurance systems have a strict requirement on correct operation. Fundamental to this is the enforcement of non-interference where particular subsystems should not affect one another. In an effort to help guarantee this policy, recent work has emerged with tracking information flows at the hardware level. This article uses a specific method known as gate-level information flow tracking (GLIFT) to provide a methodology for testing information flows in two common bus protocols, I^2C and USB. We show that the protocols do elicit unintended information flows and provide a solution based on time division multiple access (TDMA) that provably isolates devices on the bus from these flows. This paper also discusses the overheads in area and simulation time incurred by this TDMA based solution.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]

General Terms

Design, Security, Verification

Keywords

High-assurance Systems, Information Flow Tracking, Timing Channels

1. INTRODUCTION

High assurance systems such as those found in flight control and banking systems require strict guarantees on correct operation or they face catastrophic consequences. Ensuring that these systems operate as intended is an extremely difficult and costly problem. Some have estimated that such assurance can cost $10k per line of code [1] and take up to 10 years [2].

A common property that often needs to be guaranteed in these systems is non-interference [3], where certain parts of the system should never interfere with other parts. For example, the Boeing 787 aircraft has connectivity between the user and flight control networks [4]. Ensuring that there are no unintended information flows between the two networks is critical for the correct operation of the aircraft. With the further development of intricate system-on-chips interacting via complex protocols, guaranteeing non-interference is a hard problem since information can flow through difficult to detect side channels. Recently, information flow tracking (IFT) has been introduced to help mitigate this issue by monitoring how information propagates through a system.

IFT works by monitoring the propagation of data throughout a system to see if secret information is leaking to an unclassified subsystem or to ensure that the integrity of a critical subsystem is not violated by an untrusted one. There are two general classes of information flows: *explicit* and *implicit*. Explicit information flows result from two subsystems directly communicating. For example, an explicit flow would occur between a host and device on a bus that were directly exchanging data. Implicit information flows are much more subtle and generally leak information through behavior. Typical implicit information flows show up in hardware in the form of timing, where information can be extracted from the latency of operations. Previous work has shown that side channel timing attacks can be used to extract secret encryption keys from the latencies of caches [5] and branch predictors [6]. 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 on the bus [7]. Suppose there are two devices on the bus which wish to communicate implicitly. First the sender can cause excessive bus traffic when transmitting a 1 and no traffic when sending a 0. The receiver can probe the traffic on the bus to determine if the sender transmitted a 0 or 1.

To account for these difficult to detect timing channels, current methods are lacking in that they either perform

*Wei Hu is a visiting student to the Computer Science and Engineering department at UC San Diego

DAC 2011, June 5-10, 2011, San Diego, California, USA.

physical isolation or "clock fuzzing" [7, 10]. Physical isolation works by physically separating trusted/untrusted or classified/unclassified subsystems from one another. This causes overheads in area and also makes it virtually impossible to integrate subsystems together. "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.

Gate Level Information Flow Tracking (GLIFT) [11] provides a solution for monitoring information flows in hardware. Since GLIFT targets discrete gates, it is general enough to be applied to any digital hardware. Furthermore it can precisely detect all explicit information flows as well as timing channels since it monitors the change of every bit cycle-by-cycle. As a result, it is very effective for proving information flow policies about common bus protocols such as the Inter-Integrated Circuit protocol (I^2C) and the Universal Serial Bus (USB). It is essential to have methods to understand and prevent these information flows because systems, such as those found in the Boeing 787, have already begun interconnecting their high and low integrity subsystems. This paper discusses how GLIFT can be used to analyze and remove unintended information flows in bus protocols using I^2C and USB as examples.

The major contributions of this paper are:

- Presenting a method that uses GLIFT to test for unintended information flows in bus protocols that fits well with existing design flows;
- Applying this method to two common bus protocols: I^2C and USB to analyze their information flows;
- Introducing changes to the systems and discussing how these can demonstrate strong information flow isolation.

The remainder of this article is organized as follows. In Section 2 we discuss the background in hardware information flow tracking and GLIFT. Section 3 introduces the general method when analyzing information flows in bus protocols. Sections 4 and 5 discuss our method when applied to I^2C and USB respectively. We conclude in Section 6.

2. HARDWARE INFORMATION FLOW TRACKING

There has been much work in the area of hardware information flow tracking because monitoring information flows in hardware results in minimal overheads on the overall system performance. This section discusses some background on hardware information flow tracking with an emphasis on GLIFT since it is the method we used for testing information flows in I^2C and USB.

Typical hardware information flow tracking schemes tend to target the Instruction Set Architecture (ISA) and microarchitecture. Dynamic information flow tracking (DIFT), proposed by Suh et al. [8], tags information from untrusted channels and tracks it throughout a processor. They tag certain inputs to the processor as "spurious" and check whether or not this input potentially induces a control flow transfer to malicious code. Raksha [12] is a DIFT style processor that allows the flexibility of programmable security policies. Minos [9] 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 [8].

These schemes are effective at detecting dynamic information flows from spurious inputs to secret or protected regions of the architecture. However, they cannot be used to monitor information flows in general digital hardware because they target higher levels of abstraction. 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 propogate through Boolean gates. Information is said to flow through a logic gate if the inputs *affect* the output. This is done using an additional tag bit commonly referred to as *taint* and tracking logic which specifies how taint propagates.

Taint is a tag associated with each data bit in the system which indicates whether or not this particular data bit should be tracked. Taint is propogated 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.

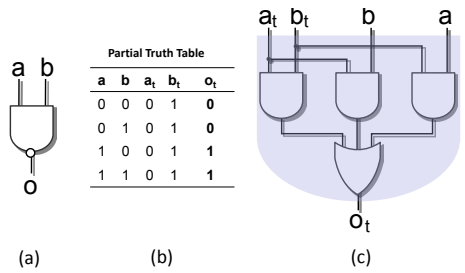

Figure 1: (a) A two-input AND gate. (b) Truth table of two-input AND gate with taint information (not all the combinations are shown). (c) The corresponding tracking logic of two-input AND gate is $ab_t + ba_t + a_tb_t$.

For example, consider a simple 2-input AND gate as seen in Figure 1 (a). For an AND 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 1 (b). Using the truth table in Figure 1 (b), a function can be derived for all similar input combinations into a tracking logic function as shown in Figure 1 (c). Following this same approach, tracking logic can be calculated for OR and NOT gates to obtain a functionally complete set of gates. Using this set, the tracking logic for any digital cirucit can be derived by constructively generating the tracking logic for each gate. This results in a design that precisely tracks all information flows.

One potential issue with GLIFT is its large hardware overhead as discussed by [15]. However, GLIFT is commonly used for testing during the design phase and does not need

to be deployed into the system. This paper focuses on using GLIFT during testing to enforce non-interference in I^2C and USB. The following section will discuss the general methodology to analyzing information flows in bus protocols using GLIFT.

3. GENERAL ANALYSIS METHODOLOGY

Since GLIFT logic targets gates, it is general enough to be applied to any digital hardware. This section discusses the test methodology when analyzing information flow properties of bus protocols using GLIFT.

As seen in Figure 2, the design typically enters as a finite state machine (FSM) modeled in RTL. This analysis is general enough to target any protocol which can be modeled as a FSM. Other RTL representations can be used, but it helps to have the design modeled as a FSM since information flow violations tend to happen on state transitions as we will show in Sections 4 and 5.

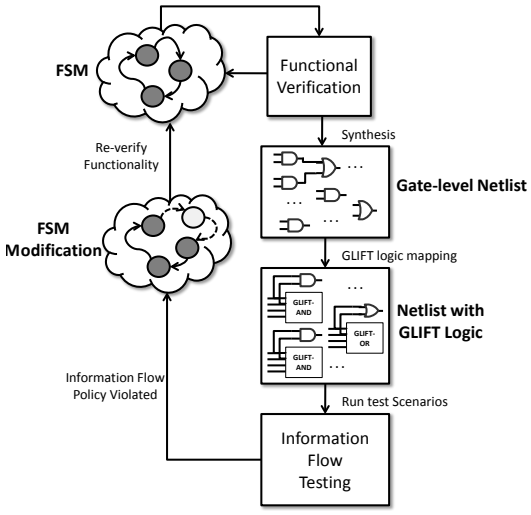

Figure 2: Method for testing information flow properties of bus protocols using GLIFT.

Once the design is synthesized into a gate-level netlist, each gate is associated with tracking logic. The function of the tracking logic depends on the function of the gate. This process is similar to a technology mapping, where each gate in the system is mapped to specific GLIFT logic. The result is a gate-level design of the FSM that contains both the original logic and tracking logic.

The resulting design equipped with tracking logic can be tested for information flows. As with any testing problem, it is often impractical to exhaustively test all states since the total grows exponentially with the number of inputs. However, GLIFT accounts for all possible combinations for tainted data bits by definition since the value of tainted data bit does not matter as proven by [15]. In our examples common and concrete scenarios are explicitly specified. Once specified, this scenario can be executed on the design and information flows can be observed. If unintended information flows occur, the designer is responsible for understanding where these originated and making appropriate modifications to the RTL. These modifications can often be subtle, but we have found some solutions to be very effective as we will show in the following sections.

4. INFORMATION FLOWS IN I^2C

This section discusses the aformentioned information flow testing technique on I^2C and shows how to obtain isolation.

I^2C is a 2-wire serial protocol consisting of a common clock and data line as shown in Figure 3 [13]. With that in mind, explicit information flows are quite simply identified since any device can openly snoop the bus even though transfer between the master and itself is never initiated. However, as this section will show, eliminating only explicit information flows in I^2C does not guarantee non-interference since information can flow through more difficult to detect side channels.

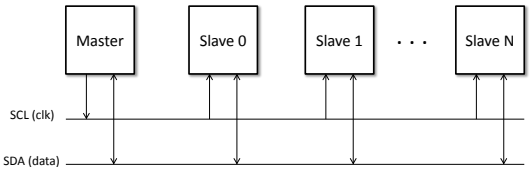

Figure 3: I^2C Bus configuration. I^2C can support several devices on a single global bus.

Our analysis of I^2C follows the same testing flow discussed in Section 3. We modeled I^2C devices as FSMs using RTL Verilog. It is not practical to enumerate all input combinations to this system since the number of states in which each device can be in grows exponentially. As a result, we chose to analyze the information flows for a common scenario in which a master writes data to a Slave 0 (as shown in Figure 3) and subsequently writes data to a Slave 1.

The design was synthesized using Synopsys Design Compiler and the gate-level functionality was also verified using Modelsim SE 6.6b. Once verified, tracking logic was associated with each gate and Slave 0 was marked as tainted because we wished to monitor where Slave 0's information flowed. During the communication between the Master and Slave 0, Slave 0 is required to send an ACK in response to receiving data. The Master's state depends explicitly on whether or not it receives an ACK. Since this ACK comes from tainted Slave 0, this ACK causes the Master's state machine to become tainted resulting in a taint explosion in the Master. As mentioned, such an explicit flow is expected since the Master and Slave 0 are directly communicating. However, once the Master subsequently communicates with Slave 1, this tainted information flows to Slave 1 resulting in a less obvious implicit information flow from Slave 0 to Slave 1. The tracking logic clearly identifies both information flows in this scenario.

In order to enforce non-interference between devices on the I^2C bus in this scenario, all information flows between slaves need to be eliminated. The following subsection discusses a useful technique for proving non-interference for this particular scenario.

4.1 Enforcing Non-interference in I^2C

To enforce non-interference between devices on the I^2C bus, we need to eliminate all information flows both explicit and implicit. This section discusses a solution for guaranteeing non-interference between devices on the I^2C bus.

As mentioned, there are obvious explicit information flows between devices since they are all connected via common wires. To eliminate explicit flows, we introduce an *adapter* which sits between the device and the bus as shown in Figure 4. This adapter arbitrates between the devices in a time division multiple access (TDMA) fashion such that only a single device is attached to the bus (in addition to the master) at any given time. In doing so, explicit information flows are eliminated since other devices are isolated from one another at all times. This does not completely eliminate all information flows (as previously mentioned) since implicit information flows between devices via the master.

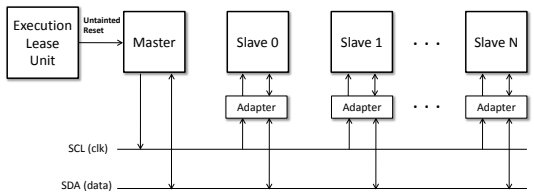

Figure 4: I²C configured with an additional adapter to enforce TDMA. This enforces non-interference between devices under the presented test conditions.

To eliminate implicit timing information flows, we introduce an untainted *reset* for the master such that the master is restored to a known state prior to communication with another device. This execution lease unit monitors when a TDMA switch occurs and restores the master to an untainted state. By requiring a strict enforcement on bus access time and by restoring the master back to a known state, we eliminate any potential timing channels between devices. Since GLIFT also captures information flows elicited through timing channels, we are able to verify that these flows are in fact eliminated for this particular scenario.

4.2 I²C Non-interference Overheads

The design was processed using the same aformentioned testing flow and synthesized with Synopsys Design Compiler. This particular scenario was tested for non-interference using Modelsim SE 6.6b. We tested a round of communication for a single time slot and verified that the information flows were in fact contained. Since information flows were proven to be contained for a time slot, information flows will be contained for this test scenario in subsequent time slots.

To obtain simulation times, we execute the complete scenario mentioned and confirmed that the master returns back to a known state without leaking any information to any devices on the bus. The simulation times for this TDMA based solution and the design with GLIFT tested are shown in Table 1. As shown, we tested this scenario with 2, 4, and 8 slaves existing on the bus. Not suprisingly, the simulation times for both the original TDMA based solution and the one with GLIFT increased with the number of slaves present on the bus. Since we are scaling the number of slaves in the system by a factor of 2, this essentially doubles the hardware and resulting simulation time. Furthermore, the overhead of the GLIFT logic does have significant effect on the simulation time relatively speaking. However, this overhead is not unwieldy since the added simulation time can likely be tolerated for such a strong information flow guarantee.

In addition, we are required to have additional hardware

Table 1: Time spent simulating our particular test scenario for both the original I²C design and the one with GLIFT.

	2 Slaves	4 Slaves	8 Slaves
TDMA Gate Design	121ms	225ms	426ms
TDMA w/ GLIFT	192ms	389ms	770ms

Table 2: Area for I²C components in non-interference compliant design. This is the final system after testing and does not contain GLIFT logic.

	Gates	Flip-Flops
Master	145	26
Slave	125	24
Adapter	375	62

(adapter) to eliminate explicit information flows between devices. Table 2 shows the sizes of each component in our testing scenario in terms of combinational and non-combinational area. It is important to note that our Master and Slave are of minimal functionality since we were only concerned with testing the previously discussed scenario. If additional complexity were added to the system (i.e., a fully functional I²C system), the Adapter's area overhead would be much less significant since its functionality is fixed (i.e., it only performs arbitration). The overhead of the execution lease unit is insignificant and not shown since it is only needed to reset the master to a known state when its timer expires. Furthermore, we are required to enforce a TDMA strategy which inherently reduces the bandwidth of the communication channel since an unused time-slot is wasted. However, this solution enforces non-interference proven by GLIFT in this particular scenario and such overheads could likely be tolerated for such a strong guarantee.

5. INFORMATION FLOWS IN USB

Unlike I²C, the Universal Serial Bus (USB) operates as a star tiered topology as shown in Figure 5. Devices are not sitting on one global bus in which explicit information may flow between one another. The Host node broadcasts data out to all Hubs and Devices. This downstream data (Host to Device) is observed by all devices and upstream data (Device to Host) is observed only by Hubs which are in the path of the stream [14]. As a result, devices are not able to snoop information sent from the Device to Host since information flows only through Hubs until it reaches the Host as shown in Figure 5. Devices can only potentially intercept information that is sent from the Host to Device since it is broadcasted. Thus the explicit information flows are less significant than in I²C, assuming USB Hubs are properly routing information. However, timing channels are still very apparent in a similar manner as I²C. This section discusses our analysis of the USB protocol along with a solution to enforce non-interference.

We are concerned with all information flows between devices. Although explicit information flows are less significant than in I²C, they still occur from the host broadcasting packets onto the USB. The less obvious types are the implicit information flows caused by state-effects on the host, i.e., a tainted device affecting the host's state. This implicit flow comes in the form of a timing channel because the amount

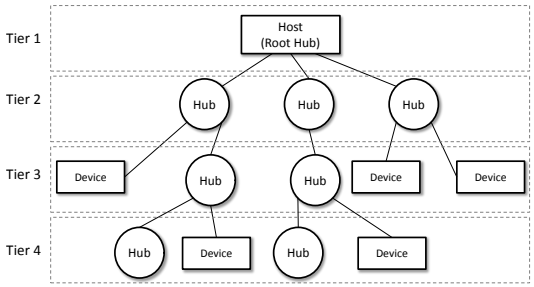

Figure 5: Packets sent from the host are broadcast onto the bus to all connected devices. The topology is a tiered star structure.

of time in which the host communicates with a device can be observed by another device. It is very similar to the implicit information flow elicited by I^2C as previously discussed.

In order to accurately model the USB protocol, we designed a USB Host and Device in RTL Verilog HDL and followed the testing flow as shown in Figure 2. These behavioral Verilog modules were functionally verified at the RTL level using Modelsim SE 6.6b. As mentioned for I^2C, testing all possible combinations is infeasible since the number of states is exponential. Thus, we have chosen a typical communication scenario consisting of 2 devices and a Host controller as shown in Figure 6. We have the Host send a packet indicating a write to Device 1 and then subsequently sends a data packet. Device 1 completes the transaction by responding with a handshake or acknowledgement packet. The Host then repeats the same procedure with Device 2.

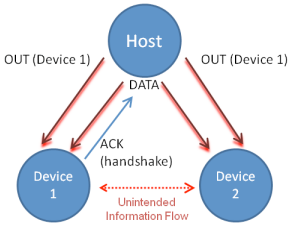

Figure 6: Host broadcasts to Device 1 and observed by Device 2. Subsequent broadcasts cause an implicit information flow between Device 1 to Device 2 through the Host.

Once the design was verified at the RTL level, we synthesized the designs to a gate-level netlist using Synopsys Design Compiler. We simulated the gate level design under the identical test conditions as those at the RTL level and verify that the circuit has functionally equivalent operations. Once verified, the gate level designs were post processed such that the tracking logic for each gate is generated. Once completed, the entire circuit has its original functionality with the addition of precise tracking logic.

The resulting gate level design with GLIFT logic was then simulated again using the same test scenario except one of the devices was labeled as *tainted* (i.e., tainted 1). In doing so, we were able to confirm that two devices, which are not physically connected, influence each other through implicit channels as shown in Figure 6. Once the host controller finishes sending packets, its state is explicitly dependent on the handshake packet received from Device 1 in a similar manner as I^2C. This information flow is captured by GLIFT and the resulting host state machine becomes tainted. Subsequently, when the Host broadcasts data, this taint is propagated to Device 2 resulting in an implicit flow between the two devices on the bus even though they are not physically on the same wire. This flow is again the result of a timing channel.

To solve this problem, we need to devise a way to reset the master back to a known state and isolate specific paths from downstream transmission. The next subsection discusses a unique TDMA solution to preventing these unintended information flows and providing isolation.

5.1 Enforcing Non-interference in USB

The TDMA solution works by modifying the Host such that it arbitrates between tainted and untainted states using a TDMA unit. In other words, the Host operates using a particular state in a fixed time slot. Once this time slot expires, the Host will switch out its state with another one (i.e., swaps out tainted for untainted). If the former was a tainted state, the switch will cause all the hardware in the Host to return to untainted. The timer allows each state machine to operate in a mutually exclusive manner. The fixed TDMA time slots prevent any timing information from flowing between the state machines since the state machines themselves have no influence on the arbitration. Conceptually, the TDMA unit in the Host acts as supervisor to the two state machines and has complete control of when tainted or untainted states can run. To account for explicit flows, the devices are tri-stated from the host when their time slot is not active to guarantee that they are not snooping on the bus.

With these additions, we synthesized the design and tested this scenario using the testing flow shown in Figure 2. We simulated our scenario for a complete time slot and verified that the information flows were contained. Again, this proves that unintended information flows are eliminated for subsequent time slots for this particular scenario. This includes information that flows through timing channels. As in I^2C, this solution results in some hardware and performance overheads and the next subsection will discuss these in more detail.

5.2 USB Non-interference Overheads

The new TDMA based solution to USB does have some minor penalties in simulation time. Yet, it results in minimal hardware overhead because the majority of the hardware does not need to be reproduced. Specifically, we are only required to have additonal logic to arbitrate between states. Once the timer expires, a new state is loaded into the Host and the old state is overwritten in a similar manner as a context switch. All internal buffers, counters, etc. remain the same.

The simulation times for the original USB design and the one equipped with GLIFT can be seen in Table 3. As in I^2C, the aformentioned test scenario was executed with 2, 4, and 8 devices on the bus. This means that we were required to replicate 2, 4, and 8 state machines respectively in the Host. This is done because a state machine is needed for each outgoing port of the Host if non-interference is to be enforced between all devices. Unlike I^2C, the simulation time does not necessarily double as devices are introduced. We suspect this is due to the fact that the majority of the hardware

Table 3: Time spent simulating the mentioned test scenario on USB with and without GLIFT.

	2 Devs	4 Devs	8 Devs
TDMA Gate Design	110ms	171ms	281ms
TDMA w/ GLIFT	187ms	297ms	531ms

Table 4: Area Overhead for Replicating State Machines

Number of FSMs	Area Overhead
2	12.6%
4	33.4%
8	77.4%
16	157.5%
32	322.9%

in the host remains the same. Thus introducing devices to the system increases the simulation time by an amount proportional to the size of the device plus a small overhead in the Host due to additional arbitration logic. Also, the simulation times for the design with GLIFT scale roughly by the same factor as the design without GLIFT. Again, we expect that the difference in time between the system with and without GLIFT to be more significant as more test scenarios are performed.

This implementation incurs a 12.6% increase in area over the original host controller for replicating a single FSM. This overhead includes the timer and logic to select between state machines. Additional FSMs are needed for each port on the Host, assuming that non-interference is to be enforced between all devices. With that in mind, the area overhead increases linearly with the increase in FSMs as shown by Table 4. These results show that much of the hardware can be re-used since the amount of overhead increases by a constant factor associated with the TDMA and extra arbitration logic. With many state machines, this overhead becomes quite large because the extra arbitration logic begins to dominate the base functional logic. The significant drawback with this design is performance. With any TDMA based scheme, performance is potentially reduced because if a device does not use its time slot when it is active, the time slot is wasted. However, such a reduction in performance can likely be tolerated at the benefit of strong information flow policies.

6. CONCLUSION

This paper presented a method for proving information flow policies in bus protocols by tracking all information flows including those through hardware timing channels. We introduced a general technique that can be used to test for information flows in bus protocols. We applied this method on two common bus protocols, I^2C and USB, and exposed unintended information flows between devices on the respective buses. We presented modifications to the systems such that they enforced non-interference between devices and proved this non-interference policy for specific test cases using GLIFT. For the modifications, we discussed what potential implications the changes have on the system in terms of simulation time and hardware overhead.

7. ACKNOWLEDGEMENTS

The authors would like to thank the reviewers for their valuable feedback which improved the final version of this paper. This work was supported by the NSF under Grant CNS-0910581.

8. REFERENCES

[1] *What does cc eal6+ mean?*, http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean/, November 20, 2008.

[2] *The integrity real-time operating system*, http://www.ghs.com/products/rtos/integrity.html, June 29, 2007

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

[4] Federal Aviation Administration (FAA). *Boeing model 787-8 airplane; Systems and Data Networks Security-isolation or Protection from Unauthorized Passenger Domain Systems Access*. http://cryptome.info/faa010208.htm.

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

[6] 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.

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

[8] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. *Secure Program Execution via Dynamic Information Flow Tracking*. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating systems (ASPLOS), New York, 2004.

[9] 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

[10] 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):1147-1165, 1991.

[11] 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.

[12] 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.

[13] *I^2C Manual*, http://www.nxp.com/documents/application_note/AN10216.pdf, March 2003.

[14] *USB 2.0 Specification*, http://www.usb.org/developers/docs, April 27, 2000.

[15] 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.