

# Tracking Information Flow at the Gate-Level for Secure Architectures

Mohit Tiwari Xun Li Hassan M G Wassel Bitu Mazloom Shashidhar Mysore  
Frederic T Chong Timothy Sherwood

Department of Computer Science, University of California, Santa Barbara  
{tiwari,xun,hwassell,betamaz,shashimc,chong,sherwood}@cs.ucsb.edu

## Abstract

*Many critical systems require tight guarantees on the flow of information, for example when handling secret cryptographic keys or critical avionics data. Unfortunately, even understanding the true flow of information through a traditional processor is difficult because executing an instruction affects so much internal state: the program counter, the memory system, forwarding and pipeline logic, and countless other bits throughout the machine. We propose a new method for constructing and analyzing architectures capable of tracking all information flow within the machine, including all explicit data transfers and all implicit flows (those subtly devious flows caused by not performing conditional operations). The key to such an approach is our novel gate-level information flow tracking method which provides a way to compose complex logical structures with well defined information flow properties. Starting from a simple NAND gate, we describe how to create more complex structures including muxes, control, registers, and finally a small microprocessor, all then implemented and tested on an FPGA. The resulting system, while less efficient than a traditional processor, is the first proof of concept demonstrating strong information-containment all the way down to the gate-level implementation.*

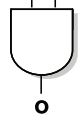
## 1 High Assurance Systems

Systems responsible for controlling aircraft, protecting the master secret keys for a bank, or regulating access to extremely sensitive commercial or military information, all demand a level of assurance far beyond the norm. Creating these systems today is an incredibly expensive operation both in terms of time and money; and even assessing the assurance of the resulting system can cost upwards of \$10,000 per line of code [2].

The enforcement of information flow policies is one of the most important aspects of such high assurance systems, yet is also one of the hardest to get correct in implementation. The recent explosion of work on dynamic dataflow tracking architectures has led to many clever

new ways of detecting everything from general code injection attacks to cross-site scripting attacks. The basic scheme keeps track of a binary property, trusted or untrusted, for every piece of data. Data from “untrusted” sources (e.g. from the network) are marked as untrusted, and the output of an instruction is marked as untrusted if any of its inputs are untrusted. While these systems will likely prove themselves useful in a variety of real-life security scenarios, precisely capturing the flow of information in a traditional microprocessor quickly leads to an explosion of untrusted state because information is leaked practically everywhere and by everything. If you are executing an exceedingly critical piece of software, for example, using your private key to sign an important message, information about that key is leaked in some form or another by almost everything that you do with it. The time it takes to perform the authentication, the elements in the cache you displace due to your operations, the paths through the code the encryption software takes, even the paths through your code that are never taken can leak information about the key.

This paper summarizes an earlier paper [7] that presented, for the first time, a processor architecture and implementation that can track *all* information-flows. On such a microprocessor it is impossible for an adversary to hide the flow of information through the design, whether that flow was intended by both parties (e.g. through a covert channel) or not (e.g. through a timing-channel). One of the key insights in this paper is that all information flows, whether implicit, covert, or explicit, look surprisingly similar at the *gate level* where weakly defined ISA descriptions give way to precise logical functions. While past approaches have assumed that any use of untrusted data should lead to an untrusted output, we observe that at the gate level this is overly conservative. If one input to an AND gate is 0, the other input can *never* affect the result and thus *should have no bearing on the trust of the output*. Based upon this observation, we introduce a novel logic discipline, *Gate-Level Information-Flow Tracking (GLIFT)* logic, which is built around a precise method for augmenting arbitrary logic blocks with tracking logic and a further method for making compositions of those blocks. Using this discipline we demonstrate how to create an architecture that, while



Logic Truth Table			Trusted A and Untrusted B				
a	b	out	a	b	a <sub>t</sub>	b <sub>t</sub>	out <sub>t</sub>
0	0	0	0	0	0	1	0
0	1	0	0	1	0	1	0
1	0	0	1	0	0	1	1
1	1	1	1	1	0	1	1

Figure 1: Tracking Information Flow through a 2-input AND Gate. Figure shows truth table for the AND Gate (left) and a part of its shadow truth table (right). The shadow truth table shows the interesting case when only one of the inputs  $a$  and  $b$  is trusted (i.e.  $a_t = 0$  and  $b_t = 1$ ). Each row of the shadow table calculates the trust value of the output ( $out_t$ ) by checking whether the untrusted input  $b$  can affect the output out. This requires checking out for both values of  $b$  in the table on the left. The gray arrows indicate the rows that have to be checked for each row on the right. For example, when  $a = 1$ ,  $b$  affects out (row 3 and 4 on the left). Hence row 3 and 4 on the right have  $out_t$  as untrusted.

unconventional in ways required by the very nature of being free from the problems of implicit-flow, is both programmable and capable of performing useful computation. We present a synthesizable processor implementation with a restricted ISA, predicated execution, bounded loops, and an iteration-coupled load/store architecture. Combined with GLIFT logic, these restrictions provide tractable and provably-sound information-flow tracking, yet allow tasks such as public-key cryptography and message authentication to be performed.

## 2 Gate Level Information Flow Tracking

Tracking all information flows through a full microprocessor is a daunting task, but one that we can tackle by breaking it down into small pieces. We begin with the smallest atomic units of logic: gates.

Consider an AND gate (shown in left side of Figure 1) with two binary inputs,  $a$  and  $b$ , and an output  $o$ . Let’s assume for right now that this is our entire system, and that the inputs to this AND gate can come from either trusted or untrusted sources, and that those inputs are marked with a bit ( $a_t$  and  $b_t$  respectively) such that a 1 indicates that the data is untrusted (or “tainted”). The basic problem of gate-level information flow tracking is to determine, given some input for  $a$  and  $b$  and their corresponding trust bits  $a_t$  and  $b_t$ , whether or not the output  $o$  is trusted (which is then added as an extra output of the function  $o_t$ ).

The assumption that most prior work makes is that when you compute a function, any function, of two inputs, then the output should be tagged as tainted if *either* of the inputs are tainted. This assumption is cer-

tainly sound (it should never lead to a case, wherein output which should not be trusted is marked as trusted) but it is over conservative in many important cases, in particular if something is known about the actual inputs to the function at runtime. To see why, let us just consider the AND gate, and all of the possible input cases. If both of the inputs are trusted, then the output should clearly be trusted. If both the inputs are untrusted, the output is again clearly untrusted. The interesting cases are when you have a mix of trusted and untrusted data. If input  $a$  is trusted and set to 1, and input  $b$  is untrusted, the output of the AND gate is always equal to the input  $b$ , which, being untrusted, means that the output should also be untrusted. However, if input  $a$  is trusted and set to 0, and input  $b$  is untrusted, the result will always be 0 regardless of the untrusted value. The untrusted value has absolutely no effect on the output and hence the output can inherit the trust of  $a$ . By including the actual values of the inputs into the determination of whether the output is trusted or not trusted, we can more precisely determine whether the output of a logic function is trusted or not.

So, how do we formalize this notion of untrusted inputs “affecting” outputs? As Figure 1 shows, essentially we are going to create a new truth table, which will *shadow* the original logic, but instead of computing the output ( $o$ ), it will compute the trust of the output ( $o_t$ ) as a function of the logical inputs ( $a$  and  $b$ ), the trust of those inputs ( $a_t$  and  $b_t$ ), and the truth table of the original function. While this seems like an awful lot of trouble to track the information flow through an AND gate, without this precision, there would be no way to restore a register to a trusted state once it has been marked untrusted. Its impact in terms of the ability to build a machine that effectively manages the flow of information is immense.

### 2.1 Composing Larger Functions

While the truth table method that we describe above is the most precise way of analyzing logic functions, our end goal is to create an entire processor using this technology. Our resulting machine is essentially going to be a large logic function which transforms a state (including the internal state of the processor, such as the program counter, and all architecturally visible state, such as the register file), to a new state based on inputs. Clearly, enumerating this entire truth table (which would have approximately  $2^{769}$  rows, where 769 is the number of state bits in our processor prototype) is not feasible, therefore we need a way of composing functions from smaller functions in a way that preserves the soundness of information flow tracking. Again, taking a smaller example to demonstrate the larger principle, let’s consider a multiplexer.

A multiplexer is small enough that we could enumerate

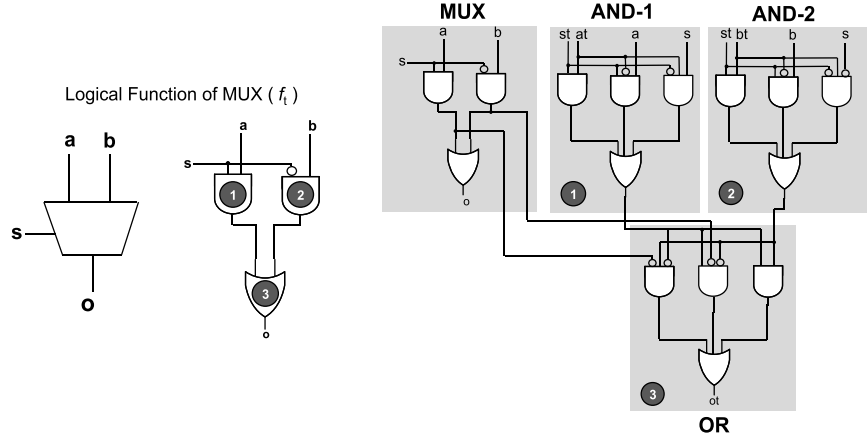


Figure 2: Composing information flow tracking logic for larger functions using basic gates. This figure shows a 2-input MUX composed of AND gates (1 and 2) and an OR gate (3). A shadow MUX is composed of shadow AND-1, shadow AND-2, and a shadow OR cells wired together the same way as the original AND1,2 and OR.

the entire function, but another way to create one is from logical gates such as AND and OR. Figure 2 shows both the logical implementation and the shadow logic. To create this shadow logic we need access to all the inputs of the MUX, and all the connections between the gates from which it is constructed. Each one of the gates from which the MUX is constructed (two AND and one OR) has a corresponding shadow logic instantiated. For example the shadow logic for ANDs (1) and (2) in the figure is simply the logic derived above. The shadow logic for OR (3), created in the same way as the AND gate, is then instantiated, and is fed the inputs from the outputs of the AND gates and the outputs of the AND shadow logic.

The shadow MUX created compositionally is, in fact, slightly more conservative than the shadow logic derived directly from the truth table. This is because the compositional approach cannot take advantage of the fact that, due to the particulars of this logic, it's impossible for the outputs of AND-1 and AND-2 to both be set to 1 at the same time, yet our OR-gate shadow logic is assuming this is possible. In this way, a compositional approach may not be exactly precise, but will always be sound and is precise enough to allow us to build useful architectures. Both capture the notion that if the select line is trusted, and the input it is selecting is trusted, the resulting output should be trusted regardless of the trustworthiness of the other input (which makes intuitive sense from an architectural perspective). The MUX, by being able to select between trusted and untrusted inputs in a way that does not propagate excessively conservatively, is the foundation of our entire architecture. For example, in Section 3.1, we will discuss how we use predication to avoid the standard implicit flow problems encountered with branches, and architecturally, predication is really a programmer-visible MUX.

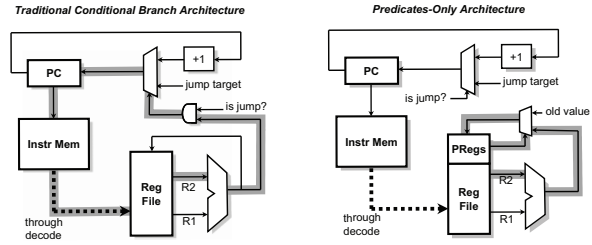


Figure 3: Implementation of a conditional branch instruction in a traditional architecture compared to ours. The highlighted wires on the left figure shows the path from an untrusted conditional to the PC. In contrast, we eliminate the path in our architecture so that the PC never gets untrusted.

### 3 Secure Architecture

After discussing the GLIFT logic method, the next question then becomes how that method can be applied to a programmable device to create an air-tight information flow tracking microprocessor. The goal of our architecture design is to create a full *implementation* that, while not terribly efficient or small, is programmable enough and precise enough in its handling of untrusted data that it is able to handle several security related tasks, while simultaneously tracking any and *all* information flows emanating from untrusted inputs.

Figure 3 shows a simple example of a branch instruction implemented in hardware and the problem with traditional architecture. Once a comparison occurs on untrusted data, the result is used to control the select line to the Program Counter, which means the PC can no longer be trusted. Once the PC is untrusted, there is no going back because each PC is dependent on the result of the last. In the architecture described above, all instructions after a branch on trusted data will be marked as

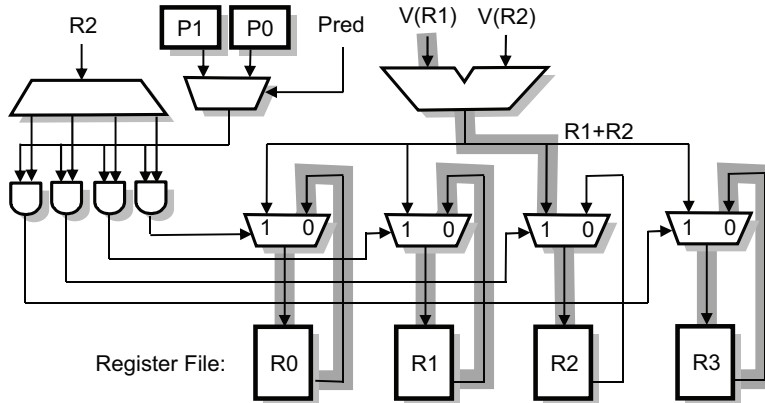


Figure 4: Implementation of our predicated architecture. The predicate bits are used to control MUXs that decide whether a register is updated with a new value or gets its old value written back into it. If the predicate bit is untrusted, the shadow MUXs ensure that all registers that could have had an untrusted value get marked as untrusted, thus turning implicit information leaks into explicitly tracked trust values.

untrusted, but is information really flowing in that way? In fact, at the gate level, it is. There is a *timing* dependence between the value of the branch predicate and the time at which the following instructions are executed. In a traditional microprocessor, there are a host of similar observable processor events whose timing is affected by some hardware logic that operates on mixed-trust data (for e.g. a bus controller) and as a result creates hidden timing channels. Such timing observations, while seemingly harmless in our example, do represent real information flow and have been used to transmit secret data [3] and reverse engineer secret keys [4].

### 3.1 Step 1: Handling Conditionals

As is apparent from our previous example, traditional conditional jumps are problematic, both because they lead to variations in timing and because information is flowing through the PC (which has many unintended consequences). Predication, by transforming if-then-else blocks into explicit data dependencies (through predicate registers), provides one answer. The effect of an instruction is guarded by a specified predicate register, and if our gate-level information flow method works correctly, the trust-bit of the destination register should be updated regardless of the value of the predicate. Since operations for both cases (predicate true/false) get executed, the augmented processor should track the information flow through every instruction that a program *could possibly* execute, even though only the instructions whose predicates evaluate to true actually write their value back to a register. As shown in Figure 3, this ensures the PC is only ever incremented, and no possible flow from untrusted input to the PC is possible.

Figure 4 shows the actual logical implementation of predication in our processor. As in a normal predicated architecture, the instruction word specifies the source registers (e.g. R1 and R2) for the instruction, destination

register (e.g R2), and a predicate register or constant (e.g P0 or P1). If the predicate register stores a 0, then the instruction doesn't write back and instead the old value is written back, but if the predicate is 1 then the new value is written. The shaded lines in the figure illustrate this point more fully. In addition to implementing predication, this example demonstrates a crucial role the MUX plays in our architecture by managing to switch between trusted and untrusted values.

### 3.2 Step 2: Handling Loops

Handling loops requires a different approach. Loops are surprisingly difficult to constrain as there are so many different ways for information to leak out in non-obvious ways. Consider a simple while-loop on an untrusted condition. Every instruction in that loop may execute an arbitrary number of times, so everything those instructions touch is untrusted. In fact, everything that *could have been modified*, even if it wasn't, needs to be marked as untrusted (due to implicit flows). To limit the effect that loops have on the untrusted state of the system, it needs to be clear, both to the programmer and at the logical implementation, exactly what state has the *possibility* of being affected by the loop. For this, we use a special `countjump` instruction that specifies statically the number of iterations that should be executed, along with the jump target for the iterations. The processor implementation then maintains a unique iteration counter for the loop instruction and ensures that the counter cannot be modified explicitly by the program. Further, `countjump` instructions have to be unpredicated, else a `countjump` would be exactly equivalent to a conditional jump and would carry all of the same problems discussed in the section above.

### 3.3 Step 3: Constraining Loads and Stores

Indirect loads and stores are another architectural feature that is problematic for information flow tracking. If a register's contents are untrusted, then using it as an address for a store instruction would implicitly mark the entire address space as untrusted (as any of those addresses could have been affected by that untrusted data). At the logical level, this shows up as the untrusted data address makes its way to the address decoder, and all of the lines of that decoder become untrusted.

Intuitively, the problem is that accessing one untrusted address causes every other address to become implicitly untrusted by virtue of them *not* being accessed or modified. To limit this implicit untrusted state explosion, in our prototype design we have limited our ISA to only support *direct* and *loop-relative* loads and stores. Direct loads use an address encoded in the immediate field, and are used to access fixed memory addresses. To allow access to arrays without resorting to general purpose indirect loads and stores, we have a loop-relative addressing mode, where loads access a variable which is at a fixed constant offset from a loop index (the loop counter used in the `countjump` instruction). This reduces convenience of programming in our ISA substantially but it allows us to precisely track any memory references. We support these by incorporating two new instructions: `load-looprel` and `store-looprel`. These are used to load and store values from a fixed base address (specified as an immediate field) and an offset stored as set of counters (C0...C7 in our prototype) that can be explicitly initialized and incremented by a fixed value using two new instructions: `init-counter` and `increment-counter`. For example, `load-looprel R0, 0x100, C0` loads the value of  $M[0x100 + C0]$  into R0. The number of times these instructions execute depends upon the number of iterations of the loop, which is fixed, and (as we did for the `countjump` instruction), the local counter initialization and increment instructions commit unconditionally so the set of all addresses that can possibly be accessed in the loop can be determined at run-time.

## 4 Evaluation

### 4.1 Implementation and Automatic Shadow Logic Generation

Our prototype processor is written in structural verilog as a composition of gates and module instantiations, along with registers and RAMs to store processor state. To augment this processor with GLIFT logic, every register gets a shadow register, every memory has a shadow RAM, and for each basic processor component like AND and OR gates, MUX-es, decoders, ALU etc, we instantiate a corresponding shadow component and wire them

up using inputs to the component and their corresponding shadow inputs.

Our processor is a 32-bit machine with 64KB each of Instruction and Data Memory, and we use Altera's QuartusII software to synthesize it onto a Stratix II FPGA. It has a program counter, 8 general purpose registers, 2 predicate registers, 8 registers to store loop counters (that count down the number of iterations) and 8 other registers to store explicit array indices (used as offsets for load-looprel and store-looprel instructions). To make the semantics of a state machine precise, all logic is triggered on the positive clock edge, and each cycle simply transforms the set of machine state into a new state through simple combinational logic. We use Altera's Nios processor as a point of comparison as it has a RISC instruction set, and, as a commercial product, is reasonably well optimized. Our base processor is almost equal in area to Nios-standard, and about double the size of Nios-economy. Adding the information flow tracking logic to the base processor increases its area by 70%, to about 1700 ALUTs.

### 4.2 Programming in the resulting ISA

To test the programmability of our design, we have hand coded a set of applications kernels onto our ISA. Our kernels are drawn from the potential program security uses of a strong information flow tracking system including a public key encryption algorithm (RSA), a block cipher (AES), a cryptographic hash (md5), along with a small finite state machine (CSMA-CD), and a sorting algorithm (bubble-sort).

Mapping applications onto our ISA requires converting conditional if-else constructs into predicated blocks, turning variable sized loops into fixed size ones (by bounding them), and turning indirect loads/stores into direct memory accesses or loop-relative ones using the loop counters. In general, any application that has predominantly regular behavior should execute without much additional overhead (e.g. RSA, md5, bubble sort), while dynamic behavior such as irregular array accesses in AES and CSMA-CD will incur much greater inefficiency. In terms of static code size, our new ISA is very close to the Nios-RISC ISA (compiled with -O2).

## 5 Conclusions

Our prototype microprocessor is in fact bigger, slower, harder to program, and computationally less powerful than a traditional microcontroller architecture. But what this architecture does for the first time is provide the ability to account for *all* information flows through the chip. It is impossible for an adversary, through clever programming, carefully crafted input, or even the use of covert or timing channels, to ever cause a resulting data

Kernel	Description	Static Instruction Count (Nios)	Static Instruction Count (this work)	Dynamic Instr. Count (Nios)	Dynamic Instr. Count (this work)	Percentage of Instr. w/ true predicates
FSM	CSMA-CD state machine with with 6 states and 4 inputs. Many table lookups	123	190	441	3322	68%
Sort	Perform bubble sort on a fixed size list of integers	26	21	20621	30358	66%
RSA	Montgomery multiplication and exponentiation from RSA public key cryptography	256	143	44880	39297	84%
AES	Block Cipher, involves extensive table lookups and complex control structures	781	1100	12807	1082207	79%
Md5	Core of the cryptographic hash function, involves mostly ALU and logical operations	769	1386	1226	1431	100%

Figure 5: A comparison of the static and dynamic instruction counts for several application kernels on our proposed ISA and an equivalent traditional RISC style architecture (the Nios). While the static instruction counts are comparable, applications that require many irregular accesses to arrays (such as indirect table look-ups) require many more instructions to select out those values.

element to be marked as “trusted” when in fact it was derived in any way from untrusted data. While covert and timing channels have been notoriously difficult to analyze in systems to date [5, 6], we capture all flows by tracking the flow of information at the level of gates, where timing signals, predicates, the bits of an address, even the internal results of logical operations all look like signals on a wire, and all of them are tracked by augmenting those structures using our GLIFT logic transformations.

We show that gate level information flow tracking, when directly applied to a traditional microprocessor, quickly points out many subtle information flows that might be hidden by the ISA abstraction, and at the very worst, lead to a quick explosion of untrusted state. We then describe an architecture that removes these problems but is still sufficiently programmable to handle a variety of small but critical tasks. Finally, through a prototype and its automatically generated information flow tracking logic, we quantify the extra area/delay cost of such flow tracking over a general-purpose microcontroller.

Over the long term we believe these techniques will find application in those critical systems like aircraft where lives are on the line. In fact, to date, no operating system has ever been rated EAL7, i.e. formally designed, tested, and verified to be provably sound from the ground up. The Integrity RTOS [1] is one of the closest at EAL6+, and even getting through the evaluation to EAL6+ required over \$10K per line of code, totaling millions of dollars over 10 years. One of the primary difficulties in getting software verified at these levels is that modern machines are simply not built with the idea of information containment in mind. This paper is hopefully a step towards reconsidering the needs of the hardware/software interface in the context of this

economically and safety critical domain.

## References

- [1] The integrity real-time operating system. <http://www.ghs.com/products/rtos/integrity.html>.
- [2] What does cc eal6+ mean? <http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean/>.
- [3] O. Aciıgmez. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture(CSAW)*, 2007.
- [4] O. Aciıgmez, J. pierre Seifert, and C. K. Koc. Predicting secret keys via branch prediction. In *The Cryptographers’ Track at the RSA Conference(CT-RSA)*, 2007.
- [5] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. ”a retrospective on the vax vmm security kernel”. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.
- [6] J. K. Millen. 20 years of covert channel modeling and analysis. In *IEEE Symposium on Security and Privacy*, 1999.
- [7] 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.