Complete Information Flow Tracking from the Gates Up

Mohit Tiwari Hassan M G Wassel Bita Mazloom Shashidhar Mysore Frederic T Chong
Timothy Sherwood

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

Abstract

For many mission-critical tasks, tight guarantees on the flow of information are desirable, for example, when handling important cryptographic keys or sensitive financial data. We present a novel architecture 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). While the problem is impossible to solve in the general case, we have created a machine that avoids the general-purpose programmability that leads to this impossibility result, yet is still programmable enough to handle a variety of critical operations such as public-key encryption and authentication. Through the application of our novel gate-level information flow tracking method, we show how all flows of information can be precisely tracked. From this foundation, we then describe how a class of architectures can be constructed, from the gates up, to completely capture all information flows and we measure the impact of doing so on the hardware implementation, the ISA, and the programmer.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-Based Systems*]

General Terms Design, Security

1. Introduction

The enforcement of information flow policies is one of the most important aspects of modern computer security, 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 through which information can be accounted for in modern software, leading to novel ways of detecting everything from general code injection attacks to cross-site scripting attacks (Dalton et al. 2007; Xu et al. 2006). 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, ultimately it is impossible for these techniques, or in fact for any security system running on a general-purpose processor, to provably capture all of the information flow within the machine (Denning and Denning 1977).

The problem is that in a traditional microprocessor, 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.

While this information leakage may not be a consideration when you are executing a word processor, leakage can be a serious problem for exceptionally sensitive financial, military, and personal data. Developers in these domains are willing to go to remarkable lengths to minimize the amount of leaked information, for example, flushing the cache before and after executing a piece of critical code (Osvik et al. 2006), attempting to scrub the branch predictor state (Aciiçmez et al. 2007), normalizing the execution time of loops by hand (Kocher 1996), and by randomizing or prioritizing the placement of data into the cache (Lee et al. 2005). While these techniques make it more difficult for an adversary to gain useful knowledge of sensitive information, at the end of the day these heuristics cannot bring the system significantly closer to a formally strong notion of information flow tracking because they do not take into consideration the intricate logic and timing that compose the implementation.

In this paper we present, 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

Copyright © 2009 ACM 978-1-60558-215-3/09/03…$5.00

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 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.

In Section 3 we describe how architectural information flows at the level of gates and present a novel compositional method by which arbitrary logic functions can be analyzed to create the fundamental building blocks of our secure hardware. In Section 4 we then describe the three major pitfalls in designing an architecture free of implicit flows, how our ISA avoids them, and how our gate-level implementation correctly tracks the resulting information flows in a provably-sound way. To ensure that the resulting architecture is not unreasonable in the additional overhead it incurs, in Section 5 we describe how this microprocessor compares with a conventional microcontroller in terms of area and performance. However, before we can begin the details of our solution, we need to begin with a discussion of the great deal of related work in both computer architecture and security that this work has built upon.

2. Related Work

The idea of tracking and constraining the flow of information is one of the primary tenets of computer security, and all manner of work has examined both the practical and theoretical limitations of mechanisms that perform this function. As has been pointed out countless times before, the general problem of determining whether information flows in a program from variable x to variable y is undecidable, as "any procedure purported to decide it could be applied to the statement **if** $f(x)$ halts **then** $y := 0$ and thus provide a solution to the halting problem for arbitrary recursive function" (Denning and Denning 1977). This is a classical example of an *implicit flow*, where information flows between variables by virtue of their *not* being accessed. For example, in the pseudo code "**if** i **then** $j := 1$", even if "$j := 1$" is *never* executed because i is always false, by observing j we can learn something about i and hence there is an information flow between i and j. If you have a Turing-complete machine, it is impossible to bound the set of possible actions that the machine might make in some conditional situation (à la the Halting Problem), and hence for any general-purpose programmable machine, it is impossible to precisely prevent *all* implicit flows. We believe our solution to this quandary is unique in that we have built a machine that, by construction, will not allow unbounded execution. In fact our design, which is still programmable through an ISA (albeit a non-traditional one), is theoretically equivalent to a single very large state machine. While this certainly limits the applicability of the machine, unbounded execution is not required to sort a bounded-size list, encrypt a message, or even verify a message signature. In the end we have created a machine in which *all* hidden flows of information are made explicit.

Using hardware to track the flow of information through a processor is by no means a new idea. DIFT (Suh et al. 2004), Minos (Crandall and Chong 2004), Rifle (Vachharajani et al. 2004), Raksha (Dalton et al. 2007), FlexiTaint (Venkataramani et al. 2008), Log-Based Lifeguards (Ruwase et al. 2008) and a host of other proposals suggest the use of data-flow tracking hardware to track the flow of untrusted network, file and user inputs through memory. The basic idea behind these tools is to assign a "tag" with every word of physical memory indicating which words of memory can be trusted, and then to track these tags around the machine as operations are performed. Every time an arithmetic operation uses an untrusted input, the output is marked as "tainted", and whenever an untrusted memory word is used for a sensitive operation like a jump address condition or a system call, the tool generates a warning for the user. Our approach, while inspired by these methods, seeks to strongly couple the notion of information flow to *all* parts of the machine at the gate level, not just the data paths, so that we know for certain that there is no way for information to be manipulated in such a way that it will "lose" the tag that represents its trust.

The idea of data-flow tracking is not limited to hardware-only options. Software projects have shown that data-flow tracking can be useful in detecting a variety of attacks (Qin et al. 2006; Costa et al. 2005; Clause et al. 2007; James Newsome and Dawn Song 2005; Xu et al. 2006; Vogt et al. 2007; Brumley et al. 2006), some with surprisingly low overhead (e.g. LIFT (Qin et al. 2006) and Speculation to Security (Chen et al. 2008)). In fact this idea can be extended to a generic taint-tracking framework that allows arbitrary policies to be enforced. Dytan (Clause et al. 2007), GIFT (Lam and cker Chiueh 2006), Taint-Enhanced Policy Enforcement (Xu et al. 2006), Raksha (Dalton et al. 2007), System Tomography (Mysore et al. 2008) and FlexiTaint (Venkataramani et al. 2008) are all examples of flexible systems for tracking data and/or enforcing polices based on those tags. In addition to explicit dataflow

tracking, some prior work has examined the problem of tracking implicit information flows (Vogt et al. 2007; Vachharajani et al. 2004; Clause et al. 2007; Xin and Zhang 2007). These approaches track information at the ISA level and attempt to combine dynamic taint tracking with limited static analysis to improve the precision of flow tracking. Our approach is different from these prior methods in that we would like to be able to precisely track all flows for any software that can be written in our ISA, and because we have knowledge of underlying hardware, we can take into consideration the logical implementation including all of its undocumented features, bugs, and timing channels.

It is worth noting explicitly what information leaks and attacks our proposed approach, taken in isolation, does not address. We do not explore the untrusted hardware component problem or physical attacks that may tamper with memory. There is already a great deal of work on tamper resistant computing (Suh et al. 2007). Nor do we consider non-digital side-channel attacks (such as those informed by observation of power distribution (Kocher et al. 1999) or RF radiation (Gandolfi et al. 2001)), as again, there are many circuit level methods for dealing with those. Instead, our approach allows us to treat the microprocessor simply as an object through which both trusted and untrusted information flows, allowing us to be certain as to which resulting outputs rely on that untrusted input. We have already begun to see mainstream processors with physically isolated protection domains, such as ARM's TrustZone (Alves and Felton 2004) and Cell Broadband Engine's Synergistic Processor Element (Shimizu et al. 2007), as a first step towards preventing trusted and untrusted data from intermingling. While, as you will see, our resulting system is not yet efficient in the traditional sense, we believe it is a leap toward the goal of a microprocessor capable of provably tracking and policing all information flows on chip.

3. 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. In this section, we begin with the smallest atomic units of logic in the microprocessor: gates. Once we precisely understand how information flows through the primitive NOT, AND, and OR gates, we can begin to compose these gates together into more complex structures such as multiplexers, arithmetic units, and eventually full processors that are able to manage and manipulate information in such a way that trust can be tracked through the implementation in a sound and precise way.

While our techniques can be extended to cover a variety of information flow security scenarios, for the purpose of this paper we will restrict our discussion to simple binary tags. Data and code are simply either "trusted" (represented logically as 0) or as "untrusted" (represented logically as 1). We have chosen a representation that is close to "taint" tracking, although we adopt the nomenclature of the security community as this is

Logic Truth Table

a	b	out
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

Trusted A and Untrusted B

a	b	a_t	b_t	out_t
0	0	0	1	**0**
0	1	0	1	**0**
1	0	0	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.

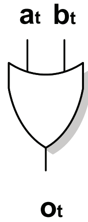
a more general information flow tracking problem rather than specifically data flow tracking. We wish to treat our whole processor as a logical function, one which operates on a set of inputs (some of which are trusted and some of which are not) and results in a set of outputs. The trust of the outputs should be determined based on the trust of the inputs, and more specifically on *how* untrusted inputs affected those outputs. To more fully illustrate the notion of trust propagation at the logical level, let's consider a very simple gate, AND. Surprisingly, even for this simple gate, the trustworthiness of the output is a complex function of the trustworthiness of the inputs *and* the actual logical values of those inputs.

3.1 Information Flow Tracking in an AND gate

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. 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).

To the best of our knowledge, all prior work in the area has assumed that if 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 certainly 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. In fact, from an information theoretic standpoint, the output of a logical function

Traditional Sound yet Conservative Trust Propagation f_t



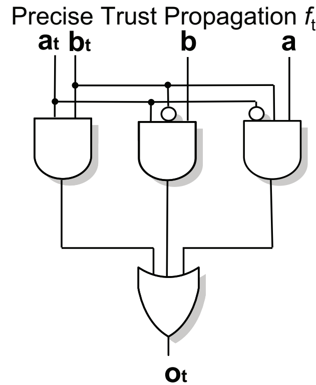


Figure 2: Shadow logic for an AND Gate. Conventional information flow tracking reports the output as untrusted if any one of the inputs is untrusted. The circuit on the right shows our shadow AND gate that marks out_t as untrusted only if out depends upon an untrusted input.

should only be untrusted if some untrusted input actually had an opportunity to affect the output[1].

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? 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 the trust of those inputs (a_t and b_t), and the truth table of the original function. Let us consider the case again where a is trusted (untrusted bit set to 0) and b is not (again in Figure 1). To compute the first line in our shadow truth table, we must consider all the possible values of the untrusted inputs (b), and

[1] while this is jumping ahead somewhat, readers familiar with implicit flows may think this sounds dangerously similar. The key difference is that we are talking about logical functions, and in a logical function it is completely possible for some inputs to have absolutely no bearing on any measurable output. The danger of implicit flows in a microprocessor is different because an action which did not happen (for example a branch of code not being taken) may result in a measurable difference of output (for example a variable not being set equal to 1).

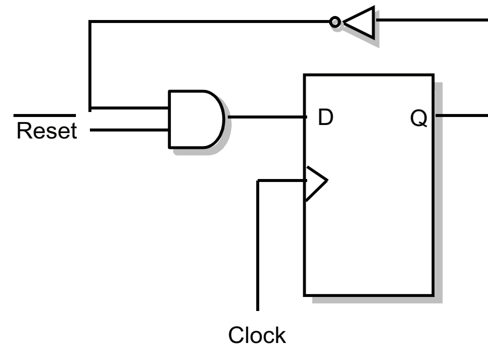


Figure 3: A 1-bit counter with reset. With the conventional technique of OR-ing all input shadow values, the feedback loop ensures that a counter shall never be trusted once it gets marked as untrusted. Our shadow logic is more precise and recognizes that a trusted reset guarantees a trusted 0 *in the counter value.*

if by changing b we can cause the output (o) to be a different value, then we know that the result cannot be trusted. For the first line of the shadow truth table, it means we need to consider the first two lines of the original truth table (the dependencies are drawn with gray arrows in the figure). Because the output is 0 for both values of b, we know that b, even if it was trying to, cannot affect the output. For the last line of the shadow truth table, we need to consider the bottom two lines of the original truth table. Because b can have an effect on the different outputs, the resulting value cannot be trusted. We can continue this process and enumerate the truth table (with 16 entries in all) for the AND gate. After minimizing to an or-of-ands representation, the resulting shadow logic is shown in Figure 2.

While this seems like an awful lot of trouble to track the information flow through an AND gate, the difference in terms of the ability to build a machine that effectively manages the flow of information is immense. Consider an extremely simple 1-bit counter that increments (or toggles in this case) every cycle, or gets cleared back to zero due to a reset. If we implement that counter as depicted in Figure 3, and use the conservative scheme from above, there is no way for that counter to ever come to a trusted state once it has been marked untrusted. However, if you use our gate-level information flow to determine the trust value, once a trusted reset has been set we know that the counter is in a trusted state 0. While this example is extremely simple, we can continue this analysis further and cover the other primitive gates and eventually analyze even the most complex of logical functions.

3.2 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 vis-

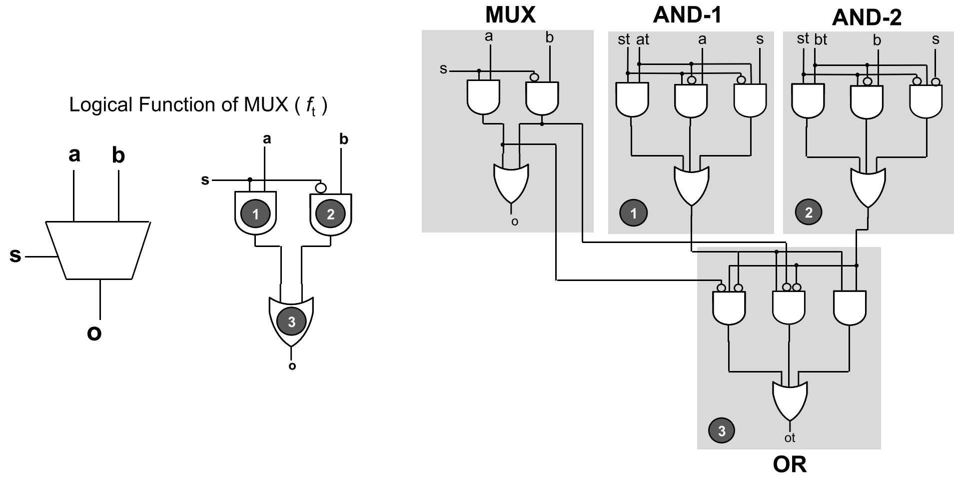


Figure 4: Composing information flow tracking logic for larger functions using basic shadow cells. Figure shows a 2-input MUX composed of AND gates (1 and 2) and an OR gate (3). A shadow MUX is composed of shadow AND-1, shadow AND-2 and a shadow OR cells wired together the same way as the original AND1,2 and OR gates.

ible 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 the entire function, but another way to create one is from logical gates such as AND and OR. Figure 4 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 in Section 3.1. 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.

One nice thing about considering a smaller example is that it is still possible to write the truth table for this example, and compare it to the result of this composition. Surprisingly, the functions are not quite identical. The shadow logic 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. In trying to calculate whether or not an untrusted input can affect output, we are essentially assuming that those uninterested inputs have more flexibility in trying to affect the output than they actually do. For our MUX example, both the precise shadow logic and the one resulting from our compositional approach are 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). Further, if the select line is untrusted, the output of the MUX will always be untrusted, except for the case when both inputs are trusted and equal. This behavior is desirable since both inputs being trusted and equal is the only case where an untrusted select cannot affect the MUX's output. More precisely, the trust value of the output of a MUX can be described by:

$$o_t = a_t s \vee b_t \bar{s} \vee s_t a_t \vee s_t a \vee s_t b_t \vee s_t b$$

In fact 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 4.1, 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.

4. Architecture

Now that we have discussed our 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.

To understand how information flows manifest themselves at the gate level, let us begin with the small snippet of pseudo

assembly below which captures nicely the notion of implicit flows discussed in Section 2. Assuming X is untrusted, should either of R1 and R2 be marked as trusted?

```
0x01 br ( X == 0 ) 0x03
0x02 R1 := 1
0x03 R2 := 1
```

Let us start with what the programmer would expect the correct answer to be: R2 does not appear to depend on the untrusted variable, and hence appears to be trusted. If $X \neq 0$ then R1 should clearly be marked as untrusted (it is set to 1 only because of a decision made on an untrusted variable). In fact, even if $X = 0$, the value of R1 is *still* dependent on X (the value of X affected value of R1 and hence there is an implicit flow).

Now let us consider what these operations would look like at the gate level on a traditional architecture that has been augmented with gate-level flow tracking. Figure 5 shows a simple example of a branch instruction implemented in hardware. The comparison occurs, and 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 our example, not only will R1 be marked as untrusted, R2 will (seemingly needlessly) be marked as well. In fact, it is even worse than that – because the PC determines the bits that set the register to writeback (and because the PC is marked as untrusted) *all* of the registers (and maybe all of memory) must also be marked as untrusted.

In the architecture described above, R2 will be marked as untrusted, but is information really flowing from X to R2? In fact, at the gate level, it is. There is a *timing* dependence between the value of X and the time at which R2 is written. Such timing observations, while seemingly harmless in our example, do represent real information flow and have been used to transmit secret data (Aciiçgmez 2007) and reverse engineer secret keys (Aciiçmez et al. 2007).

Modern processors are simply not built to constrain information flow. Rather, they are built to get things done as quickly as possible, often times making use of as much information as possible at each step to make that happen. Our approach to the problem is to restrict both the ISA of the machine and the actual gate level implementation so much that, a) all information flow will be obvious and well understood at the assembly level, b) the actual propagation of trust-bits corresponds closely with this understanding, c) it is impossible to write programs that will result in “explosions” of untrusted state, d) the information flow will be precisely tracked no matter what binary is given to the machine (there is no compiler pre-analysis step required to ensure the strength of information flow tracking) e) it is always possible to return the machine to a trusted state, and f) the shadow information-flow-tracking logic can be composed and added automatically in the way described in Section 3 resulting in the tracking of *all* information flows (implicit, timing, covert, or otherwise).

The resulting processor looks like a large state machine, where the state is defined by the architectural and internal state of the processor (PC, flags, registers, counters, etc.), and an arbitrarily large but *finite* amount of memory (a subtle but important distinction). Given the current state at cycle i, you simply compute the next state for cycle $i+1$. In the subsections below we describe several devious ways in which information will flow through a machine in ways the programmer is not intending, and the architecture changes required to avoid them.

4.1 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). Removing conditionals presents a challenge: how to provide conditional operations without modifying the PC? Predication, by transforming if–then–else blocks into explicit data dependencies (through predicate registers), provides an 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 5, this ensures the PC is only ever incremented, and no possible flow from untrusted input to the PC is possible.

Figure 6 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. Let us consider the following predicated code, and how trust-bits would flow through the logic in this example.

```
0x01 ( 1)  P1 := not( P0 )
0x02 (P0)  R2 := R1 + R2
0x03 (P1)  R0 := R1 + R2
```

In this code, either of R0 or R2 gets the sum $(R1 + R2)$ written into it (based upon the conditional P0). Let us consider what happens to the architecture pictured in Figure 6 on instruction 0x02 if P0 is untrusted. First, the untrusted predicate will be

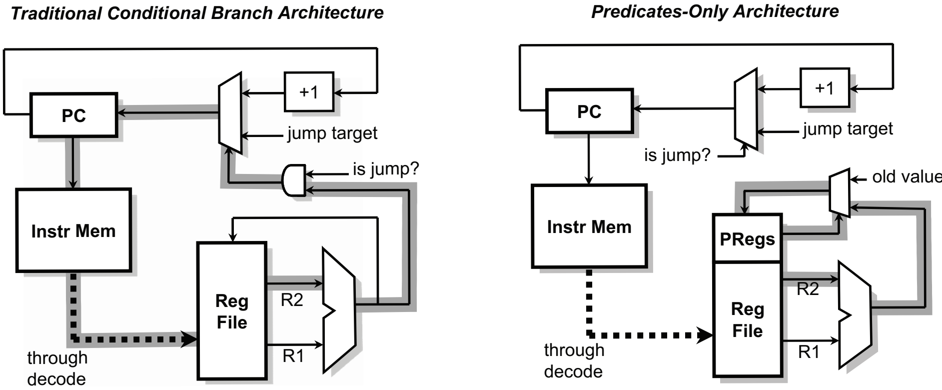


Figure 5: Implementation of a conditional branch instruction in a traditional architecture compared to ours. The highlighted wires on the left figure shows the path from an untrusted conditional to the PC. In contrast, we eliminate the path in our architecture so that the PC never gets untrusted.

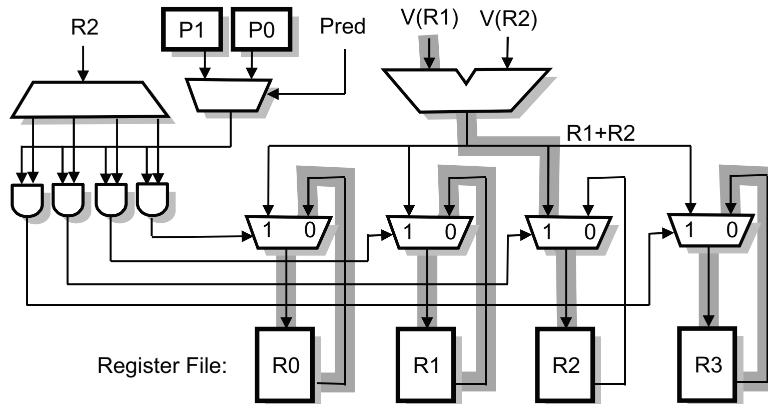


Figure 6: Implementation of our predicated architecture. The predicate bits are used to control MUXs that decide whether a register is updated with a new value or gets its old value written back into it. If the predicate bit is untrusted, the shadow MUXs ensure that all registers that could *have had an untrusted value get marked as untrusted, thus turning implicit information leaks into explicitly tracked trust values.*

selected by the MUX, and will be used (in conjunction with R2) to select the register to write back (this is happening at the bank of small AND gates). As the number 2 flows through the decoder, all of those lines feeding the AND gates *except for the one line controlling* R2 will be set to 0. For each of those lines, the untrusted predicate is now irrelevant because we can trust that output of the AND gate will be 0 no matter what (as per our discussion of AND gates earlier in the paper), hence, the values on those lines can be trusted. For the one remaining line (the one controlling R2), one of the inputs to the AND gate is 1, while the other input in untrusted, and hence the result on that line must be untrusted *no matter whether the predicate is true or false*. That untrusted line will then control the final MUX that determines if the new value or old value should be written back, which will result in R2 being marked as untrusted (again, regardless of the predicate being true or false).

As a programmer, this complex interplay between the original logic and the information tracking logic is actually quite intuitive. If you predicate an instruction on an untrusted predicate, the destination register will be marked as untrusted. It is as simple as that. As an architect, once you manage to eliminate the spurious information flows, the automated methods described in Section 3 actually manage to augment the logic in a way that is both sound and in-line with programmer expectations.

4.1.1 Step 2: Handling Loops

While we can use predication to eliminate the use of conditional jumps in the case of if-then-else blocks, 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. Consider a loop with i going from 0 to X, and setting A[i] := 1. The fact that A[$X+1$] = 0 tells us something about X, and so there is information flow from X to A[$X+1$]. In fact there is information flow from X to A[$X+n$] for all n less than the maximum possible value X can ever have. Even the fact that the loop may take an arbitrary number of cycles creates an implicit timing channel with all of the instructions downstream from it.

To limit the effect that loops have on the untrusted state of the system, we have to severely constrain the types of loops that are possible in the system by bounding the side-effects that a loop can have. 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. While predication makes the side effects of conditional operation explicit, to deal with loops 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.

Counting loop instructions have existed in the context of DSPs for some time, but we believe this is the first time they are being used to aid information tracking. The `countjump` instruction has three interesting details. First, `countjump` has to be unpredicated, implying that it will always commit and a constant amount of jumps to the jump target will always be performed. If `countjump` were to be predicated, it would be exactly equivalent to a conditional jump and would carry all of the same problems discussed in the section above. Second, it is supported by an internal counter that gets set the first time the instruction is encountered. On all subsequent executions, the counter decrements by 1 until it reaches 0. One further execution will find the counter at 0 and advance the PC by 1 to exit the loop instead of jumping to the jump target. The third detail is that, in order to support nested loops, if a dynamic instruction instance finds the counter at 0, then it gets reset back to the specified value and the entire loop is restarted. This functionality is implemented by an internal state machine that sets the counter back to an uninitialized state when the counter is found to be 0 and the loop is found to be terminated. By using predicates inside the loops, a programmer can simulate an "early termination" by predicating all instructions in the loop body with the negation of this termination condition. In Section 5.2 we discuss the ramifications of this on the programmer in a bit more detail.

4.1.2 Step 3: Constraining Loads and Stores

The example for loops above also demonstrates a third architecture feature that is problematic for information flow tracking: indirect loads and stores. Most ISAs support indirect memory addressing, where a register's contents provides the address for a load or a store. If the 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.

4.2 Implementation and Automatic Shadow Logic Generation

Our prototype processor is implemented in Verilog, and we use Altera's QuartusII software to synthesize it onto a Stratix II FPGA. It is a 32-bit machine with 64KB each of Instruction and Data Memory. 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. This logic uses the PC to read out an instruction word, decode it, perform data memory accesses and ALU computations and finally write back new values into registers, memory and PC. In practice, block RAMs in Stratix II FPGAs are synchronous and require two cycles to read data out. Our simple processor executes an instruction every 5 cycles similar to the classical 5-stage multicycle machine. We have avoided the complications of pipelining (especially the forwarding logic it requires) for the purposes of this proof-of-concept.

Our 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 we proceed in two steps. First, each bit of processor state is explicitly shadowed, meaning every register gets a shadow register, and every memory has a shadow RAM (that

stores the 1-bit trust values for each bit of the orginal memory). Second, the logic and signals are shadowed by generating the proper trust propagation logic as described in Section 3.

The first step is easily accomplished by simply duplicating the declarations for registers and memory. To handle the second step we create a library of shadow cells that perform information flow tracking for each basic processor component like AND and OR gates, MUX-es, decoders, ALU etc. The shadow logic is wired up with both the inputs to the original function, and also with corresponding shadow inputs. While we could spend time describing more formally how this happens, it is easiest to simply see from the resulting verilog code (Figure 7).

4.2.1 Programming in the resulting ISA

Figure 8 summarizes our instruction set. We eliminate conditional jumps and indirect loads and stores from our ISA, and introduce a countjump instruction to execute fixed-size loops, predicated instructions to implement conditional execution, and restricted loads and stores that use only immediate values. In addition to these instructions, we support various logical (AND, OR, NOT, XOR), arithmetic (ADD, SUB), bitwise (SHR, SHL) and comparison operators. As an example usage of the new instructions, let us consider a code snippet from the `SubBytes` function in the GLIFT implementation of the AES (Daemen and Rijmen 2002) encryption algorithm (in Figure 9). The function substitutes values in the `state` matrix with values in the `SBox`. The code below loads the value in the `state` matrix (which in this example is stored starting at address 0x100). Every loaded element serves as an index into the `SBox`, and is substituted by the value in the `SBox` (which is stored starting at address 0x300). The `state` has 16 elements and the `SBox` is a 256 entry table, correspondingly, the `countjump` instructions 0x0b and 0x08 loop back a fixed number of times (15 and 255 respectively).

5. Evaluation

To demonstrate that our proposed architecture is actually implementable, we have built a working model of our processor on an FPGA, and we have written several application kernels to help us quantify the overheads involved. Figure 10 shows one portion of that result, the area and frequency overhead of our proposed architecture, both with and without GLIFT logic added, as compared to a NIOS processor.

5.1 Hardware Impact

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. The Nios can be instantiated as either an economy core (Nios-econ) or a standard core (Nios-std). The economy version is an unpipelined 6 stage multicycle processor without caches, branch-predictors etc. (most closely comparable with our core), while the standard version is pipelined and has an additional 4KB instruction cache. The area and timing numbers have been generated by synthesizing the GLIFT-base (with no information flow tracking logic), GLIFT-full and Altera-Nios processors onto a Stratix II device with compilation settings balancing optimization for both area and delay. In Figure 10, the left Y-axis shows the area required to implement the processors measured in Adaptive Look-Up Table (ALUT) units (the logic cells used by Altera Stratix II FPGAs), while the right Y-axis shows the maximum frequency (Fmax) of the processors.

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. However, in terms of absolute size, even the now outdated Stratix II FPGAs have upto 180K ALUTs, while all the above processors consume only in the range of 1K-2K ALUTs. On the right Y axis, Fmax for Altera Nios processors is around 160MHz, while both the base and full GLIFT processors have an Fmax of around 130MHz. In terms of delay, both GLIFT and Nios are multi-cycle processors with the path through the ALU to the destination register being the most critical. The extra tracking logic does not impose a significant overhead on the Fmax, reducing it from 131MHz to 129MHz. Further, the GLIFT processors operate at 130 MHz as opposed to 160 MHz because we include a barrel shift that the Nios does not (with 1-b shifts, our processor also operates at 160MHz.). While these overheads are certainly non-trivial, keep in mind that this version of the processor shadows *every* bit in the machine. By trading off precision for efficiency it may be possible to keep the soundness of our result while reducing the performance impact.

5.2 Analysis of Application Kernels

To test the programmability of our design, we have hand coded a set of applications kernels onto our ISA. This allows us to examine the impact of our modified ISA on the the static code size and the dynamic instruction count of the programs. 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, while dynamic behavior such as irregular array accesses will incur much greater inefficiency. For our experiments, we implemented each of the programs under test both directly in our assembly and in C. The C programs are compiled down to Nios-RISC executables with "-O2" and emulated with Altera's instruction set simulator (ISS). Our assembly is mapped to our FPGA implemen-

Original Logic | **Added Logic**

Original Logic	Added Logic
reg [31:0] gen_reg [7:0];	reg [31:0] gen_reg_shadow [7:0];
wire [31:0] mux2greg0;	wire [31:0] mux2greg0_shadow;
always @ (posedge clk) begin	
g_reg[0] <= mux2greg0;	g_reg_shadow[0] <= mux2greg0_shadow;
end	
assign is_store = instrn[29] | instrn[22];	assign is_store_shadow = (instrn_shadow[29] & Instrn_shadow[22])
	| (instrn_shadow[29] & (~ instrn_shadow [22]) & (~ instrn [22]))
	| ((~ instrn_shadow[29]) & instrn_shadow[22] & (~ instrn[29]));
mux2x1_32b my_mux0(.in0(g_reg0),	mux2x1_32b_shadow sh_my_mux0(.in0(g_reg0), .in0_t(g_reg0_shadow),
.in1(newval), .sel(p_sel0),	.in1(newval), .in1_t(newval_shadow), .sel(p_sel0), .sel_t(p_sel0_shadow),
.result(mux2greg0));	.ot(mux2greg0_shadow));

Figure 7: An example of how our very structured verilog code can be automatically augmented with the logic required to track the trust through the hardware implementation. Each wire, register, and signal is augmented with a corresponding shadow element that stores the 1-bit trust value for each.

Instruction	Pred	Action	Information Flow
load-immediate	yes	Rdest := immed	Rdest inherits the trust of the predicate
load-direct	yes	Rdest := M[immed]	Rdest is truted if both the predicate and the memory value are trusted
store-direct	yes	M[immed] := Rsrc	The memory value is trusted if the predicate and Rsrc are trusted
load-looprel	yes	Rdest := M[immed + LCount]	Rdest is trusted if the memory value and the predicate are trusted
store-looprel	yes	M[immed + LCount] := Rdest	The memory values is trusted if the predicate and Rdest are trusted
add, sub, and, or, not, xor, shl, shr, cmpeq, cmplt	yes	Standard 3-address register to register operations	Rdest is trusted if the both the inputs to the ALU operation are trusted
predset	yes	Pdest := Rsrc	Pdest is trusted if the predicate and Rsrc are trusted
countjump	no	Jump to target exactly N times (N specified in immediate field)	The loopcounter can only be written by an immediate and should never become untrusted
init-counter	no	LCount := 0	LCount is trusted
increment-counter	no	LCount := LCount + 1	LCount remains trusted

Figure 8: An overview of the ISA of our prototype architecture, and the information flow tracking policies that are extracted from the actual logic level implementation.

```
0x01  ( 1) load-immediate     P1 := 0              #
0x02  ( 1) init-counter       C0 := 0              # i = 0
0x03  ( 1) load-looprel       R0 := M[0x100 + C0]  # R0 = state[i]
0x04  ( 1) init-counter       C1 := 0              # j = 0
0x05  ( 1) cmpeq              P1 := C1, R0         # if ( j == R0)
0x06  (P1) load-looprel       R1 := M[0x300 + C1]  #    R1 = SBox[j]
0x07  ( 1) increment-counter  C1 := 1              #    j++
0x08  ( 1) countjump          (0x05), 255          # loop back 255 times
0x09  ( 1) store-looprel      M[0x100 + C0] := R1  # state[i] = R1
0x0a  ( 1) increment-counter  C0 := 1              # i++
0x0b  ( 1) countjump          (0x03), 15           # loop back 15 times
```

Figure 9: Example usage of the new GLIFT instructions: a code snippet from the `SubBytes` *function in the GLIFT implementation of the AES (Daemen and Rijmen 2002) encryption algorithm.*

tation to ensure the correctness of our design, and is then run through our instruction set simulator to gather dynamic instruction counts. Figure 11 presents the results of those experiments.

In terms of static code size, our new ISA is very close to the Nios-RISC ISA. However, the dynamic instruction counts vary substantially. Programs such as the CSMA-CD finite state machine and AES have numerous table look-ups where each look-up requires a full table iteration. As a result, these have a very large dynamic instruction count in comparison to the general purpose ISA. On the other hand, bubble sort, which also requires array accesses, is fairly efficient because both the Nios and our ISA loop over the entire array N^2 times. Any inefficiency there is owing to instructions that were executed but not written back because their predicates were false. Finally, RSA and md5 have very little in the way of predicated instructions, and both comprise mainly of ALU instructions. For these

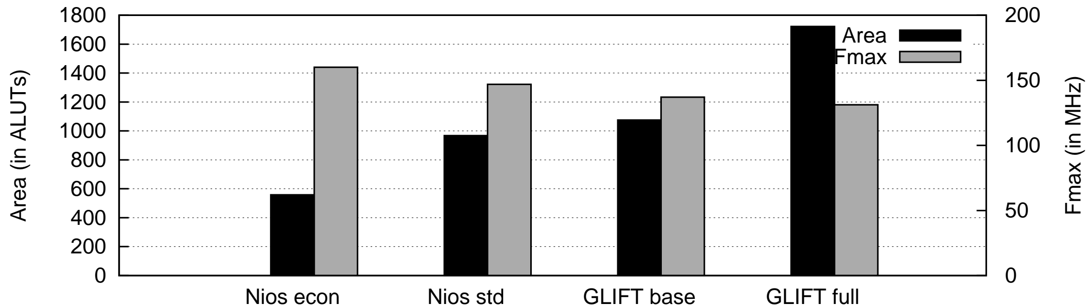


Figure 10: Quantifying the area and timing overhead of gate-level information flow tracking. The left Y-axis compares the number of FPGA logic cells required to implement a basic GLIFT processor (which implements the ISA but doesn't include the shadow logic) and a full information flow tracking GLIFT processor to two general purpose micro-processors by Altera.

applications, the number of executed instructions is very close to the Nios. While our assembly is unoptimized and opportunities for optimization abound, the main point of this is to show that it is indeed possible to constrain a processor enough that all information flow is apparent at the gate level, yet still maintain enough programmability that programs can be mapped to it without an unmanageable amount of overhead.

6. Conclusions

At the end of the day, our new microprocessor is 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 element to be marked as "trusted" when in fact it was derived in any way from untrusted data. This is accomplished 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. When critical or sensitive operations need to be performed, a co-processor augmented with these abilities could be an attractive option.

We devise a flow tracking logic for simple gates that considers the *effect* of inputs on outputs while propagating taint directly from the truth tables of those gates, and propose a sound composition rule to generate shadow logic for more complex structures. We then 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 go on to describe an architecture that removes these problems while still retaining sufficient programmability to allow it to handle a variety of small but critical tasks. Finally, by implementing a prototype and automatically augmenting it with our information flow tracking logic, we quantify the extra area/delay cost of such flow tracking over a general-purpose micro-controller. While there are many opportunities to further optimize both our architecture and our application kernels, the techniques presented here show that is indeed possible to track information flows through a programmable design.

Acknowledgments

The authors would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF Career Grant CCF-0448654, and by grants FA9550-07-1-0532 (AFOSR MURI) and NSF 0627749.

References

James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

Onur Aciiçgmez. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture(CSAW)*, 2007.

Onur Aciiçmez, Jean pierre Seifert, and Cetin Kaya Koc. Predicting secret keys via branch prediction. In *The Cryptographers' Track at the RSA Conference(CT-RSA)*, 2007.

Tiago Alves and Don Felton. TrustZone: Integrated Hardware and Software Security, July 2004. URL `http://www.arm.com/products/esd/trustzone_home.html`.

David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

Haibo Chen, Xi Wu, Liwei Yuan, Binyu Zang, Pen chung Yew, and Frederic T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. *Intl. Symposium on Computer Architecture (ISCA)*, 2008.

James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, 2007.

Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating Systems Principles(SOSP)*, 2005.

Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of*

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 11: 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.

the 37th annual IEEE/ACM International Symposium on Microarchitecture(MICRO), 2004.

Joan Daemen and Vincent Rijmen. The design of rijndael: Aes - the advanced encryption standard. 2002.

Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *34th Intl. Symposium on Computer Architecture (ISCA)*, June 2007.

Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7), 1977.

Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and Embedded Systems*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer-Verlag, 2001.

Paul Kocher, Joshua Ja E, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology*, pages 388–397. Springer-Verlag, 1999.

Paul C. Kocher. Timing attacks on implementations of die-hellman, rsa, dss, and other systems. pages 104–113. Springer-Verlag, 1996.

Lap Chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference(ACSAC)*, 2006.

Ruby B. Lee, Peter C. S. Kwan, John P. Mcgregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005.

Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. Understanding and Visualizing Full Systems with Data Flow Tomography. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2008.

Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2006.

Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting General Security Attacks. In *Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.

Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures(SPAA)*, 2008.

K. Shimizu, H. P. Hofstee, and J. S. Liberty. Cell broadband engine processor vault security architecture. *IBM J. Res. Dev.*, 51(5):521–528, 2007. ISSN 0018-8646.

G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th international conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2004.

G.E. Suh, C.W. O'Donnell, and S. Devadas. Aegis: A single-chip secure processor. *Design and Test of Computers, IEEE*, 24(6):570–580, Nov.-Dec. 2007. ISSN 0740-7475.

Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Fourteenth International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *ISSTA*, pages 185–195, 2007.

Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, 2006.