Gate-Level Information-Flow Tracking for Secure Architectures

This article describes a new method for constructing and analyzing architectures that can track all information flows within a processor, including explicit, implicit, and timing flows. The key to this approach is a novel gate-level information-flow-tracking method that provides a way to create complex logical structures with well-defined information-flow properties.

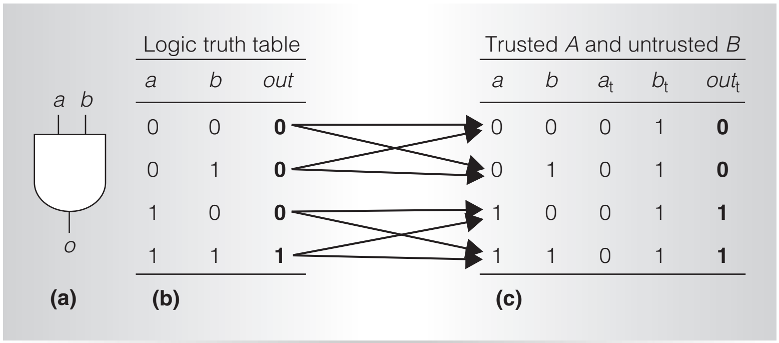
Mohit Tiwari
Xun Li
Hassan M.G. Wassel
Bita Mazloom
Shashidhar Mysore
Frederic T. Chong
Timothy Sherwood
University of California, Santa Barbara

Systems responsible for controlling aircraft, protecting master secret keys for a bank, or regulating access to extremely sensitive commercial or military information all demand an assurance level far beyond the norm. These strict demands for high-assurance systems stem directly from the cost of failure. For example, Boeing plans to use a single physical network for both aircraft control data and passenger network data in its new 787; protecting the integrity of the critical data from all other data is obviously important.

Similarly, if a secret such as a private key is necessary to perform an operation, an attacker might be able to reverse-engineer the key via direct-timing observations or cache interference,[1] or even through the branch predictor's state.[2] Once an attack has been identified and publicized, effective countermeasures can be deployed, such as randomizing the cache's replacement policy,[3] but this constant attack-and-respond cycle is inadequate when the cost of leaking data (for example, a bank's root private key or a military authorization code) is extraordinarily high. Creating such systems today is an incredibly expensive operation, in terms of both time and money. Even assessing the resulting system's assurance can cost more than $10,000 per line of code.[4]

Interestingly, protecting either integrity or confidentiality depends on the ability to track information as it flows through a program. The enforcement of information-flow policies is one of the most important aspects of such high-assurance systems, yet it is also one of the hardest to get right during implementation. A considerable amount of recent 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.[5] The basic scheme keeps track of a binary property, trusted or untrusted, for every piece of data. Data from untrusted sources (for instance, the network) are marked as untrusted, and an instruction's output is marked as untrusted if any of its inputs are untrusted.

Although such systems will likely prove useful in various real-life security scenarios, precisely capturing the information flow in a traditional microprocessor quickly leads to a 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—almost everything you do with that key leaks information about it in some form. Information about the key can



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 two-input AND gate (a), with truth table (b) and shadow truth table (c). The shadow truth table shows the case when only one of the inputs, *a* or *b*, is trusted (for example, $a_t = 0$ and $b_t = 1$). Each row of the shadow table calculates the trust value of 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 of (b). The arrows indicate the rows that must be checked for each row in (c). For example, when $a = 1$, *b* affects *out*, as rows 3 and 4 in (b) show. Therefore, rows 3 and 4 in (c) list out_t as untrusted.

leak because of the time it takes to perform the authentication, the elements in the cache you displace because of your operations, the paths through the code that the encryption software takes, and even the paths through your code that are never taken.

In this article, we describe a processor architecture and implementation that can track all information flows.[6] In such a microprocessor, it is impossible for an adversary to hide the information flow through the design, whether that flow was intended by both parties (for instance, through a covert channel) or not (for instance, through a timing channel). One of the key insights in this work is that all information flows—whether implicit, covert, or explicit—look surprisingly similar at the gate level, where weakly defined instruction-set architecture (ISA) descriptions give way to precise logical functions.

Previous approaches have assumed that any use of untrusted data should lead to an untrusted output, but 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 output's trust level. On the basis of this observation, we introduce a novel logic discipline called *gate-level information-flow tracking* (Glift), which we 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, although 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 that combines predicated execution and bounded loops, along with space- and time-bounded function calls. Combined with Glift logic, these restrictions provide tractable, provably sound information-flow tracking, while also supporting tasks such as public-key cryptography and message authentication.

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 with two binary inputs *a* and *b* and output *o* (see Figure 1a). Assume for now that this is our entire system, 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), where a 1 indicates 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 output *o* is trusted (which is then added as an extra output of the function o_t).

Most prior work assumes that when you compute any function of two inputs, the output should be tagged as tainted if either input is tainted. This assumption is certainly sound (it should never lead to a case in which an output that should not be trusted is marked as trusted), but it is too conservative in many important cases—especially if something is known about the actual inputs to the function at runtime. To see why, consider the AND gate and all possible input cases. If both inputs are trusted, the output should clearly be trusted. If both inputs are untrusted, the output is again clearly untrusted. The interesting cases are when there's a mix of trusted and untrusted data. If input *a* is trusted and set

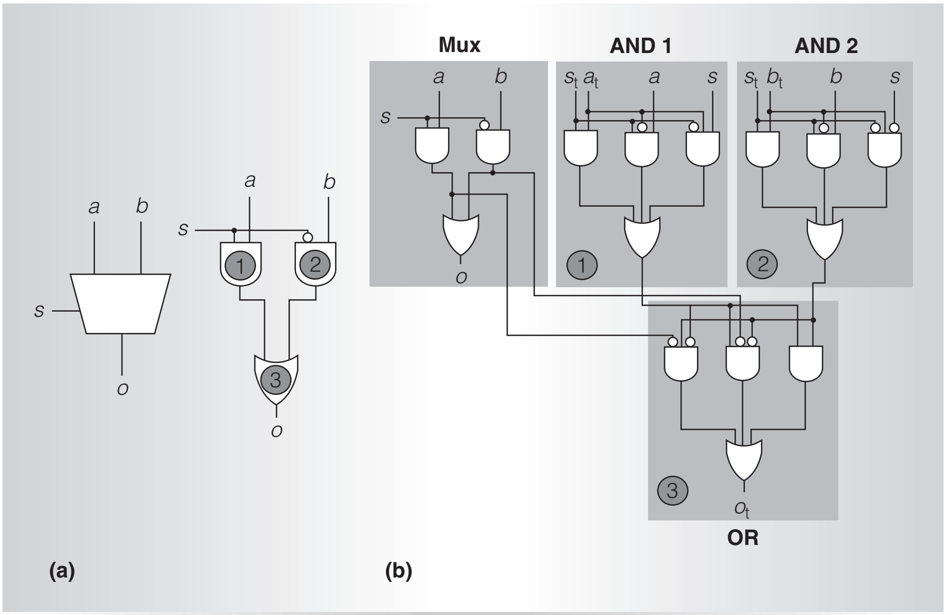


Figure 2. Composing information-flow-tracking logic for larger functions using basic shadow cells: The logical function f_t of a two-input multiplexer (with input *s* selecting between inputs *a* and *b*) includes two AND gates (1 and 2) and an OR gate (3) (a). A shadow multiplexer consists of shadow AND 1, shadow AND 2, and a shadow OR cells wired together the same way as the original AND 1, AND 2, and OR gates (b).

to 1, and input *b* is untrusted, then the AND gate's output is always equal to input *b*; because *b* is untrusted, 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 inputs' actual values into the determination of whether or not the output is trusted, we can more precisely determine whether or not a logic function's output is trusted.

So, how do we formalize this notion of untrusted inputs affecting outputs? As Figure 1 shows, we essentially create a new truth table that *shadows* the original logic. But, rather than the output *o*, this new truth table computes the output's trust o_t as a function of the logical inputs *a* and *b*, those inputs' trust a_t and b_t, and the original function's truth table. Although tracking the information flow through an AND gate seems like an awful lot of trouble, without this precision there would be no way to restore a register to a trusted state once it was marked untrusted. The impact of this approach on the ability to build a machine that effectively manages the information flow is immense.

Composing larger functions

Although the truth table method just described is the most precise way to analyze logic functions, we aim to create an entire processor using this technology. The resulting machine will essentially be a large logic function that transforms a state, including the processor's internal state (such as the program counter) and the entire 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 that preserves the soundness of information-flow tracking. Again, taking a smaller example to demonstrate the larger principle, we consider a multiplexer.

A multiplexer (mux) is small enough that we could enumerate the entire function, but another way to create a mux is from logical gates such as AND and OR. Figure 2 shows

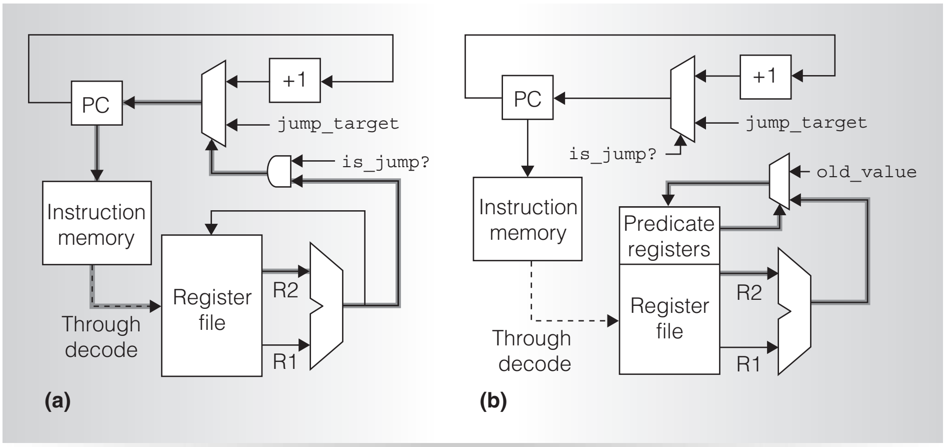


Figure 3. Implementation of a conditional-branch instruction in a traditional architecture (a) compared to our predicated architecture (b). The highlighted wires in (a) show the path from an untrusted register to the program counter (PC). In contrast, we eliminate the path in our architecture so that the PC never becomes untrusted.

both the logical implementation and shadow logic for such a mux. To create this shadow logic, we need access to all mux inputs, as well as all the connections between the gates from which the mux is constructed. Each of these gates (two AND and one OR) has a corresponding shadow logic instantiated. For example, the shadow logic for AND 1 and AND 2 in Figure 2 is simply the logic derived earlier. The shadow logic for the OR gate, created in the same way as for each AND gate, is then instantiated and fed the inputs from the AND gates' outputs and the AND shadow logic's outputs.

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 exploit the fact that this mux's logic makes it impossible to simultaneously set both the AND 1 and AND 2 outputs to 1, yet our OR gate shadow logic assumes this is possible. Thus, a compositional approach might not be exactly precise, but it will always be sound, and it is precise enough for us to build useful architectures. Both the shadow mux created compositionally and the shadow logic derived directly from the truth table capture the notion that if the *select* input of the mux and the input that the mux chooses are trusted, the resulting output should be trusted regardless of the other input's trustworthiness (which makes sense intuitively from an architectural perspective). The mux, by being able to choose between trusted and untrusted inputs without propagating the tainted data too conservatively, is the foundation of our entire architecture.

Secure architecture

Now that we've discussed the Glift logic method, we must address how to apply this method to a programmable device to create an air-tight information-flow-tracking microprocessor. Our architecture design aims to create a full implementation that is programmable and precise enough in its handling of untrusted data to deal with several security-related tasks, while simultaneously tracking all information flows emanating from untrusted inputs; we aren't concerned right now about efficiency or size.

Figure 3 shows a simple example of a branch instruction implemented in hardware and highlights the problem with traditional architectures. Once such an architecture performs a comparison using untrusted data and uses the result to control the *select* line to the program counter, the program counter can no longer be trusted. Once the program counter is untrusted, there is no going back, because each value of the program counter depends on the previous value.

In the architecture just described, all instructions after a branch on untrusted

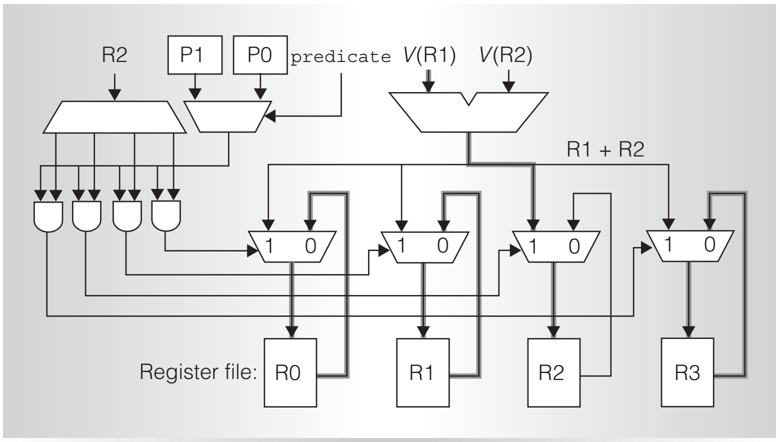


Figure 4. Implementation of our predicated architecture. The predicate bits control muxes that decide whether a register is updated with a new value or gets its old value written back to it. If the predicate bit is untrusted, the shadow muxes ensure that all registers that could have had an untrusted value get marked as untrusted, thus turning implicit information leaks into explicitly tracked trust values. [*V*(R1) and *V*(R2) are the values of registers R1 and R2, respectively.]

data (and consequently the entire CPU state) are marked as untrusted. But is information really flowing that way? In fact, at the gate level, it is. The path taken after the conditional-jump instruction explicitly affects the contents of memory addresses that are targets of store instructions. Furthermore, the path that is not taken implicitly affects the contents of addresses that don't contain values that they could potentially have. In fact, there is also a timing dependence between the branch predicate's value and the time at which the following instructions are executed. In a traditional microprocessor, the timing of many observable processor events are affected by some hardware logic operating on mixed-trust data (for example, a bus controller), creating hidden timing channels. Such implicit flows and timing observations, although seemingly harmless in our example, represent real information flow and have been used to transmit secret data and to reverse-engineer secret keys.[1,2]

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 flows through the program counter (which has many unintended consequences). Predication, by transforming if-then-else blocks into explicit data dependencies through predicate registers, provides one solution. A specified predicate register guards an instruction's effect, and if our gate-level information flow method works correctly, the destination register's trust bit should be updated regardless of the predicate's value. Because operations for both cases (predicate true or false) are 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 Figure 3 shows, this ensures that the program counter is only incremented and not assigned the target of a conditional jump, and hence no information can flow from untrusted inputs to the program counter.

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 (R1 and R2) for the instruction, the destination register (R2), and a predicate register or constant (P0 or P1). If the predicate register stores a 0, the instruction doesn't write back and, instead, the old value is written back. But if the predicate is 1, 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 switching between trusted and untrusted values.

Step 2: Handling loops

Handling loops require a different approach. Loops are surprisingly difficult to constrain because information can leak out in so many unexpected ways. Consider a simple *while* loop on an untrusted condition. Every instruction in that loop could 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, must be marked as untrusted (because of implicit flows). Limiting the effect that loops have on the system's untrusted state requires making it clear to both the programmer and the logical implementation exactly what state the loop could possibly affect. Therefore, 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 program cannot explicitly modify the counter. Moreover, `countjump` instructions must be unpredicated, or else a `countjump` would be exactly equivalent to a conditional jump and would carry all the same problems discussed in Step 1.

Step 3: Constraining loads and stores

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

To limit this implicit untrusted-state explosion, the basic Glift ISA supports only direct and loop-relative loads and stores. Direct loads use an address encoded in the immediate field, and they are used to access fixed memory addresses. A loop-relative addressing mode allows access to arrays. In this mode, loads access a program variable at a fixed constant offset from a loop index (the loop counter used in the `countjump` instruction). However, programming with only direct memory accesses and predicating all conditions causes the static code size to bloat and performance to suffer badly. To overcome this problem, we introduce the concept of *execution leases*.

Step 4: Executing general-purpose code

Execution leases are architectural mechanisms that let an untrusted entity control a portion of the machine for a fixed amount of time and within a fixed range of addresses.[7] Within a lease, the untrusted code can execute conditional jumps and indirect memory accesses. The gate-level implementation of the lease CPU guarantees that no effect of the general-purpose code escapes the lease bounds. After the lease expires, control is yanked back to the trusted code, and any remnants of the untrusted actions are purged from the critical machine state, such as the PC. (Registers and main memory are not part of the critical machine state, and so they retain their values and security labels even after a lease expires.)

Execution leases thus lift the restrictions on indirect memory accesses and conditional-jump instructions, and they allow code within lease bounds to execute with the performance of general-purpose code. We implement execution leases with special instructions, `settimer` and `setbounds`, which provide a new function-call mechanism at the language level: `lease(timer, memorysize, Function(arg0,...), returnvalue)`, which allows calls to `Function`, with `timer` and `memorysize` as the time and memory bounds. (These bounds can be decided with the help of our prototype compiler.[7])

Implementation and shadow-logic generation

Our prototype processor is written in structural Verilog as a composition of gates and module instantiations, along with registers and RAM to store processor state. To augment this processor with Glift logic, we create a shadow register for every register, and a shadow RAM for every memory. We also instantiate a corresponding shadow component for each basic processor component, such as AND and OR gates, muxes, decoders, and the ALU, and we compose the shadow components together using inputs to the components and their corresponding shadow inputs.

Our processor is a 32-bit machine, with 64 Kbytes each of instruction memory and data memory, and we used Altera's Quartus II software to synthesize it onto a Stratix II FPGA. It has a program counter, eight general-purpose registers, two predicate registers, eight registers to store loop counters (that count down the number of iterations), and eight other registers to store explicit array indices (used as offsets for `load-looprel` and `store-looprel` instructions). To make the semantics of a state

Table 1. Comparison of static- and dynamic-instruction counts for a traditional RISC processor (the Altera Nios), our original microcontroller based on gate-level information-flow tracking (Glift), and our processor updated with execution leases.

		Static instructions			Dynamic instructions		
Kernel	**Description**	**Nios**	**Original Glift**	**Execution leases**	**Nios**	**Original Glift**	**Execution leases**
FSM	Carrier Sense Multiple Action with Collision Detection (CSMA-CD) finite-state machine with six states and four inputs, and many table lookups	123	190	130	441	3,322	410
Bsort	Performs bubble sort on a fixed-size list of integers	26	21	126	20,621	30,358	43,518
RSA	Montgomery multiplication and exponentiation from RSA public-key cryptography	256	143	95	44,880	39,297	26,329
AES	Advanced Encryption Standard block cipher involving extensive table lookups and complex control structures	781	1,100	2,113	12,807	1,082,207	15,785
MD5	Message-Digest algorithm 5 core of the cryptographic hash function involving mostly ALU and logic operations	769	1,386	951	1,226	1,431	1,012
MM	Matrix multiplication	108	73	83	9,043	17,035	10,094

machine precise, we trigger all logic on the positive clock edge, and each cycle simply transforms each machine state into a new state through simple combinational logic. We compare our processor to Altera's Nios embedded processor because the latter has a RISC instruction set, and, as a commercial product, is reasonably well-optimized. Our base processor is almost equal in area to the Nios standard processor and about double that of the Nios economy processor. Adding the information-flow-tracking logic to the base processor increases its area by 70 percent, to about 1,700 adaptive lookup tables (ALUTs).

Programming in the resulting ISA

To test our design's programmability, we hand-coded a set of application kernels onto our ISA. We chose these kernels from potential program-security uses of a strong information-flow-tracking system, including a public-key encryption algorithm (RSA), a block cipher (Advanced Encryption Standard, or AES), a cryptographic hash (Message-Digest algorithm 5, or MD5), a small finite-state machine (Carrier Sense Multiple Action with Collision Detection, or CSMA-CD), and a sorting algorithm (bubble sort).

Mapping applications onto our ISA requires converting conditional if-else constructs into predicated blocks, turning variable-size loops into fixed-size ones (by bounding them), and turning indirect loads and stores into direct memory accesses or loop-relative ones using loop counters. In general, any application that has predominantly regular behavior (for example, RSA, MD5, and bubble sort) should execute without much additional overhead, whereas dynamic behavior such as irregular array accesses in AES and CSMA-CD will incur far greater inefficiency in the basic Glift ISA. This inefficiency is mitigated in the execution lease ISA by allowing bounded, general-purpose code. In terms of static code size, our new ISA is very close to the Nios RISC ISA (compiled with the –O2 optimization level).

Table 1 compares Altera's Nios RISC processor, our basic Glift-based microcontroller, and our processor updated with execution leases. Although the static instruction counts are comparable, applications needing many irregular accesses to arrays (such as indirect table lookups in finite-state machines and AES) require many more instructions to predicate out those values in the basic Glift ISA. With execution leases, a CPU can execute indirect memory accesses securely, and the dynamic instruction counts become comparable to a general-purpose Nios ISA.

Our prototype microprocessor is bigger, slower, harder to program, and computationally less powerful than a traditional microcontroller architecture. But it is the first architecture that can account for all information flows in a chip. This approach makes it impossible for an adversary—through clever programming, carefully crafted inputs, or even covert or timing channels—to cause a resulting data element to be marked as trusted if it was derived from untrusted data. Although covert and timing channels have been notoriously difficult to analyze,[8,9] we capture all information flows by tracking them at the gate level, where timing signals, predicates, address bits, and even the internal results of logic operations all look like signals on a wire. Our method tracks all of them by augmenting those structures using our Glift logic transformations.

In the long term, the techniques we've described in this article will find application in critical systems such as avionics, where lives are on the line. Thus far, no operating system has ever been rated EAL7 (meaning formally designed, tested, and verified to be provably sound from the ground up). Integrity's Real Time Operating System is one of the closest at EAL6+,[10] but even getting that far cost more than \$10,000 per line of code, totaling millions of dollars over 10 years. One of the primary difficulties in verifying software at these levels is that modern machines are simply not built with the idea of information containment in mind. This article is hopefully a step toward reconsidering the needs of the hardware-software interface in the context of this economically and safety-critical domain. MICRO

References

1. D.A. Osvik, A. Shamir, and E. Tromer, ''Cache Attacks and Countermeasures: The Case of AES,'' *Proc. Topics in Cryptology* (CT-RSA 06), LNCS 3860, Springer, 2006, pp. 1-20.
2. O. Aciiçmez, C.K. Koç, and J.-P. Seifert, ''Predicting Secret Keys via Branch Prediction,'' *Proc. Topics in Cryptology* (CT-RSA 07), LNCS 4377, Springer, 2007, pp. 225-242.
3. R.B. Lee et al., ''Architecture for Protecting Critical Secrets in Microprocessors,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 2-13.
4. ''What Does CC EAL6+ Mean?'' Open Kernel Labs Blog, 20 Nov. 2008; http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean.
5. M. Dalton, H. Kannan, and C. Kozyrakis, ''Raksha: A Flexible Information Flow Architecture for Software Security,'' *Proc. 34th Ann. Int'l Symp. Computer Architecture* (ISCA 07), ACM Press, 2007, pp. 482-493.
6. M. Tiwari et al., ''Complete Information Flow Tracking from the Gates Up,'' *Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 09), ACM Press, 2009, pp. 109-120.
7. M. Tiwari et al., ''Execution Leases: A Hardware-Supported Mechanism for Enforcing Strong Non-interference,'' *Proc. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 42), IEEE CS Press, 2009, pp. 493-504.
8. P.A. Karger et al., ''A Retrospective on the VAX VMM Security Kernel,'' *IEEE Trans. Software Eng.,* vol. 17, no. 11, 1991, pp. 1147-1165.
9. J.K. Millen, ''20 Years of Covert Channel Modeling and Analysis,'' *Proc. IEEE Symp. Security and Privacy* (SP 99), IEEE CS Press, 1999, pp. 113-114.
10. Integrity Real-Time Operating System, Green Hills Software, 2009; http://www.ghs.com/products/rtos/integrity.html.

Mohit Tiwari is pursuing a PhD in the Department of Computer Science at the University of California, Santa Barbara. His research interests include computer architecture and program analysis, especially for security and debugging. He has a BTech in computer science from the Indian Institute of Technology, Guwahati.

Xun Li is pursuing a PhD in the Department of Computer Science at the University of California, Santa Barbara. His research interests include computer architecture, computer security, and compilers. He has a BSc in software engineering, jointly from Fudan University (China) and University College Dublin.

Hassan M.G. Wassel is pursuing a PhD in the Department of Computer Science at the University of California, Santa Barbara. His research focuses on computer architecture and its interaction with software and novel hardware technologies. He has an MSc in computer science and automatic control from Alexandria University in Egypt. He is a student member of the IEEE.

Bita Mazloom is pursuing a PhD in computer science from the University of California, Santa Barbara. Her research interests include dataflow analysis, information-flow modeling, reverse engineering of software, visualization of systems, and pedagogical techniques in computer science. She has a BS in computer science from the University of California, Santa Barbara.

Shashidhar Mysore is a product specialist at Eucalyptus Systems. He completed the work described in this article while pursuing his PhD at the University of California, Santa Barbara. His technical interests include developing technological expertise and strategies for innovation based on cutting-edge research and extensive market analysis in cloud computing, and understanding complex computer systems. He has a PhD in computer science from the University of California, Santa Barbara. He is a member of the IEEE.

Frederic T. Chong is the director of computer engineering and a professor of computer science at the University of California, Santa Barbara. He also directs the Greenscale Center for Energy-Efficient Computing. His research interests include emerging technologies for computing, multicore and embedded architectures, computer security, and sustainable computing. He has a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology.

Timothy Sherwood is an associate professor of computer science and electrical engineering at the University of California, Santa Barbara. His research interests include software profiling and debugging peripherals, novel uses of 3D integration, whiteboard-based sketch computing, MEMS (microelectromechanical systems) control architectures, and secure hardware-software systems. He has a PhD in computer science and engineering from the University of California, San Diego.

Direct questions and comments to Mohit Tiwari, Dept. of Computer Science, University of California, Santa Barbara, Office 1119, Harold Frank Hall, Santa Barbara, CA 93106-5110; tiwari@cs.ucsb.edu.