Hardware Implementation of Information Flow Signatures Derived via Program Analysis

Paul Dabrowski, William Healey, Karthik Pattabiraman, Shelley Chen[§], Zbigniew Kalbarczyk, and Ravishankar K. Iyer

Center for Reliable and High Performance Computing,
University of Illinois (Urbana-Champaign)
{pdabrows, whealey, pattabir, kalbar, rkiyer}@uiuc.edu

[§]*SAIC*
Champaign, IL
shelley.chen@saic.com

***Abstract* — We present an architectural solution that provides trustworthy execution of C code that computes critical data, in spite of potential hardware and software vulnerabilities. The technique uses both static compiler-based analysis to generate a signature for an application, or operating system, and dynamic hardware/software signature checking. A prototype implementation of the hardware on a soft processor within an FPGA incurs no performance overhead and about 4% chip area overhead, while the software portion of the technique adds between 1% and 69% performance overhead in our test applications, depending on the selection of critical data.**

I. Introduction

With shrinking process technology driving an increase in the number of transistors per chip, processor designers are utilizing these transistors in more creative ways. While new innovations are now possible, more complicated and tedious validation stages in the design cycle are introduced. For example, the complex interactions of multiple cores on a chip, new levels of abstraction such as virtualization extensions, and huge growth in the size of the hardware description language code contribute to more processor bugs slipping by validation engineers than ever before [1]. The Intel Core 2 Centrino architecture has an 82 page document dedicated just to workarounds for over 100 processor errata which have been uncovered thus far [2]. Though no known working exploits for these errata have been released, the list grows every month and has raised lively discussion about the security implications of processor bugs [3][4]. As with each new processor release, the errata list is extended. It is easy to foresee a future where an operating system not implementing workarounds is susceptible to both these known and other yet to be discovered low-level vulnerabilities.

Furthermore, these are only examples of unintentional implementation bugs due to insufficient validation. Unfortunately, it is also possible for a rogue designer to intentionally introduce hard-to-find bugs deep inside a processor, with the intention of being able to exploit the unknown hardware errors once the chip is being produced and distributed on a large scale. This topic has been a recent focus of several large initiatives [5], and has garnered significant attention in the research community [6].

Both of these situations lead to very real security vulnerabilities that challenge traditional thinking in computer security. It is generally considered that by erecting a "virtual fence" that protects the computer system, those with malicious intent can be kept out. For example, current software-based security techniques tend to focus on remote vulnerabilities, such as buffer overflow attacks, format string attacks, and other memory corruption attacks, looking to either "plug" every potential hole in the software [7][8][9], or to make sure the statistical likelihood of a successful attack is extremely low [10][11][12]. Several hardware-based runtime security techniques have also emerged to prevent similar classes of attacks [13][14]. These techniques improve upon the performance overheads of software-based techniques. However, they still fall short in being able to provide guarantees if an attacker is already in the system. The fact that an attacker bypassed the virtual fence in a way not protected by these techniques, for example, a potential processor bug, means that the virtual fence not useful at this point. In the face of vulnerabilities that utilize processor bugs, it is worthwhile to consider security techniques that operate under the assumption that a "hole" in the virtual fence may be found at any time.

This paper presents a hardware implemented enforcement of "Information Flow Signatures" extracted via program analysis. The enforcement ensures that, during runtime, security critical data is computed according to source code semantics, even under the threat of hardware or software vulnerabilities. If an attacker attempts to tamper with the protected program execution, the system will be notified so that it can take appropriate actions. Though this technique can also be used to provide protection against a wide range of memory corruption attacks, we focus specifically on the hardware-based mechanism for protecting security critical data against malicious tampering. The specific contributions

This work was supported in part by the U.S. Department of Commerce under Grant SBAHQ-05-I-0062, NSF grants CRI CNS 05-51665, CNS 05-24695, CNS 04-06351, Gigascale Research Center (GSRC/Marco), Hewlett Packard, Intel Corporation, and Motorola Corporation.

of this study are:

1. Design, prototype implementation and demonstration of Information Flow Signatures (IFS) checking hardware for trustworthy execution of instructions computing critical data. The key advantages of the IFS technique include:
   - Small hardware footprint
   - Does not affect processor clock frequency
2. Automated compiler-based application analysis for deriving Information Flow Signatures, and application instrumentation to facilitate communication between the application and IFS checking hardware.

II. Information Flow Signatures For Data Integrity

The Information Flow Signatures (IFS) technique is used to protect the integrity of critical data within an application or operating system [15]. This data may be selected as the highest priority security variables, such as variables that hold information about user-authentication in an SSH application or structures containing information about currently running processes within an operating system.

Once the developer selects the critical data, extraction of the Information Flow Signatures takes place as a compiler pass that requires no further programmer intervention. The backwards slice of instructions and data that can directly or indirectly influence the critical data is encoded and added to the final binary as part of the program initialization. During runtime, the Information Flow Signatures is checked against the executing instructions, detecting any abnormalities, which can then be handled by code also protected by the technique itself.

A. Assumptions and Threat Model

In general, the IFS technique described here can be used to prevent data corruption attacks. The aim of the technique is to preserve data integrity rather than its confidentiality. Hence, the technique does not address side-channel attacks [16]. The threat model assumes that the attacker can execute arbitrary code and overwrite program variables stored in both memory and processor registers. We assume all malicious memory accesses are visible to the processor pipeline. Thus, malicious DMA transfers are *not* covered by this threat model. For example, an attacker could use an IEEE 1394 interface port to initiate transfers to the main memory of a system while remaining unnoticed by the processor [17]. We also assume that the Information Flow Signature hardware is initialized prior to the attack conditions; thus, the technique protects against runtime attacks, but only if the correct signatures have been loaded into the hardware. We have explored solutions to correctly initializing the hardware under the threat of insider attacks or an untrusted operating system in [18], but do not consider these conditions here.

Table 1: Runtime actions for the Level 1 hardware check

Destination / (I.pc, I.operands)	Critical	Non-Critical
(Trusted, Trusted)	Invoke Level 2 Check	Invoke Level 2 Check, set trusted bit of target
(Trusted, Non-Trusted)	Raise Alarm	Raise Alarm
(Non-Trusted, Trusted)	Raise Alarm	Clear trusted bit of target
(Non-Trusted, Non-Trusted)	Raise Alarm	Clear trusted bit of target

B. Compiler Analysis and Runtime Checking

The protection scheme consists of two phases:

1) A compile-time phase to extract the backward slice of critical data in the program, and

2) A runtime-phase to check if the critical data is influenced is in violation of the statically derived backward slice. The runtime phase is implemented using a combination of software and hardware.

Compile-time Phase. Given code instrumented with annotations denoting critical data, compiler-based static analysis determines the following:

1) Instructions that can influence the critical data (according to program semantics), and

2) The set of objects (data) that each instruction (in (1)) is allowed to write to

The compiler generates an Information Flow Signature that encodes each instruction in (1) and each object in (2). All instructions in this Information Flow Signature, and the data at the beginning of the backwards slice, are marked *trusted*. This *trusted* property is propagated at runtime according to the propagation rules for each executed instruction and its operands as shown in Table 1. An example backwards slice used to derive the Information Flow Signature is shown in Figure 1, where the data at address 0xC004 has been marked critical.

Runtime Phase. The following invariants are enforced at runtime using a combination of hardware and software:

1) *Level 1 Check*: Critical data is modified only by *trusted* instructions and objects (enforced in hardware).

2) *Level 2 Check*: Each *trusted* instruction writes only to its statically allowed objects, as defined in the source code and derived by the compiler analysis (enforced in software)

As the *trusted* property of variables is propagated for all instructions and data at runtime, any instruction which was originally not in the backwards slice of the critical data that tries to influence the critical data, will cause the *trusted* property of critical data to be overwritten. The MOV instruction on the left of Figure 1 shows an example of this unsuccessful attack. Since the Level 1 hardware checker will catch this malicious attempt, an attacker must now subvert an

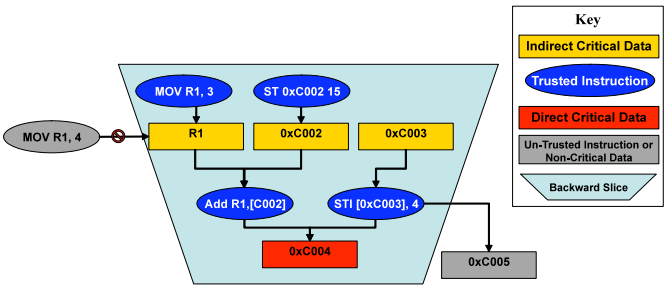

Figure 1: An Example Backwards Slice

instruction that is trusted into writing to an object other than the one it is supposed to. However, this is protected through the Level 2 software checking. It is called on all trusted instructions storing data to memory. If either the Level 1 or 2 checks fail, an interrupt is triggered which halts execution of the program and raises a security alert. This interrupt routine can itself be protected using the Information Flow Signatures technique. By raising an alert immediately on detecting a write to the slice, the integrity of critical data can be ensured, and allows for immediate action to be taken by the application or operating system.

For further discussion and resolution of issues regarding the effects of conservative compiler analysis in deriving the backwards slice, user-input being part of the backwards slice, and dynamically mapping trusted data refer to [15].

III. Hardware Implementation

The Information Flow Signature technique hardware checking mechanism is implemented as a Reliability and Security Engine module, as described below.

A. The Reliability and Security Engine

The Reliability and Security Engine (RSE) is a framework that provides a standard interface between a processor pipeline and hardware modules that implement reliability and security services for the executing application [19]. Figure 2 illustrates a block diagram of the RSE connected to the Gaisler Research Leon 3 open-source VHDL processor pipeline [20].

The modules are running alongside the host processor, monitoring the behavior of the executing application. Probes inserted into the pipeline of the host continuously transfer select host state information to the RSE modules. Reconfigurable hardware slices similar to those in an FPGA may be used to implement the modules, allowing for instantiation of the desired modules based on the requirements of the current application. Alternatively, modules can be implemented as an ASIC IP core, which are imported by a processor designer to suit the demands imposed by the customers and the market.

In this design, we override the SPARC v8 instruction set architecture co-processor operation instruction (CPOP1), converting it into a CHK instruction. This CHK instruction is used for communication between an instrumented application and the RSE modules. It is ignored by the main pipeline of the processor, and considered a NO-OP. The CHK instructions are uniquely identified to specify the module they are intended for. For example, during application initialization, the Information Flow Signatures technique uses CHK instructions to convey the signatures of the trusted instructions and critical data that are required to enforce the Level 1 checks in hardware.

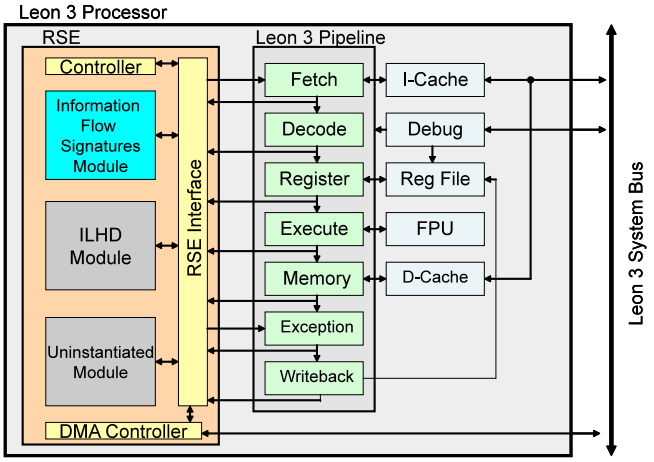

Figure 2: The Leon 3 processor with the RSE Interface

The RSE also contains a DMA controller connected to the system bus in parallel with the processor. With the RSE, module designers need not worry about making direct changes to a processor pipeline. This allows one to focus on the performance of only the components being added, rather than their interaction with the rest of the processor. Additionally, the RSE minimizes the intrusiveness of developing new techniques, which decreases the chances of RSE module designers from introducing errors that affect the functionality of the chip in unexpected ways.

B. Information Flow Signatures Checking Module

To propagate the trusted and critical information associated with instructions and data, the IFSCM implements a pipeline structure similar to that in the main processor. By using the RSE interface for all required pipeline control information, no changes to the microprocessor pipeline are required for the Information Flow Signatures Checking Module (IFSCM). We use a data storage structure named "Critical Data/Trusted Instruction" (CDTI) within the IFSCM to track when this information is written to memory, and thus do not require any modifications to the processor caches or system busses. In order to alleviate any storage limitations that may arise from a fixed-size CDTI, we also propose modifications to the TLB that allow for unlimited trusted instructions and data in Section V.A. However, the currently implemented fixed-sized CDTI is shown to be adequate for the medium-sized applications we have examined.

The Level 1 check described in Section II.B is executed in hardware by the IFSCM. This guarantees that the check has a low performance overhead and that it is performed on every executed instruction. The Level 2 check is performed using a software interrupt routine, which is only called when the Level 1 check finds a trusted instruction is writing to memory.

A prototype of the IFSCM is implemented in Field Programmable Gate Array (FPGA) hardware interfacing with the Leon 3 open-source VHDL processor. The signals read by the IFSCM from the Leon 3's RSE interface include: 1) the register file control, 2) the current instruction and its pointer, 3) an indicator for pipeline stalls, flushes, and 4) the cache control. These signals are used directly from the processor's pipeline without modifications. Figure 3 shows the checking module. It contains a pipelined structure similar to the main processor's pipeline, and a small register file to track intermediate trusted data before it is written to the CDTI. Signals read from the processor pipeline are used to control this checking pipeline. Outputs from the checking module trigger an interrupt within the processor, allowing the software to handle Level 2 checks and security violations.

The CDTI. Using the CDTI to store the trusted and critical bits obviates the need to use system RAM to mark instructions and data as trusted or critical. Thus, we relieve the need to

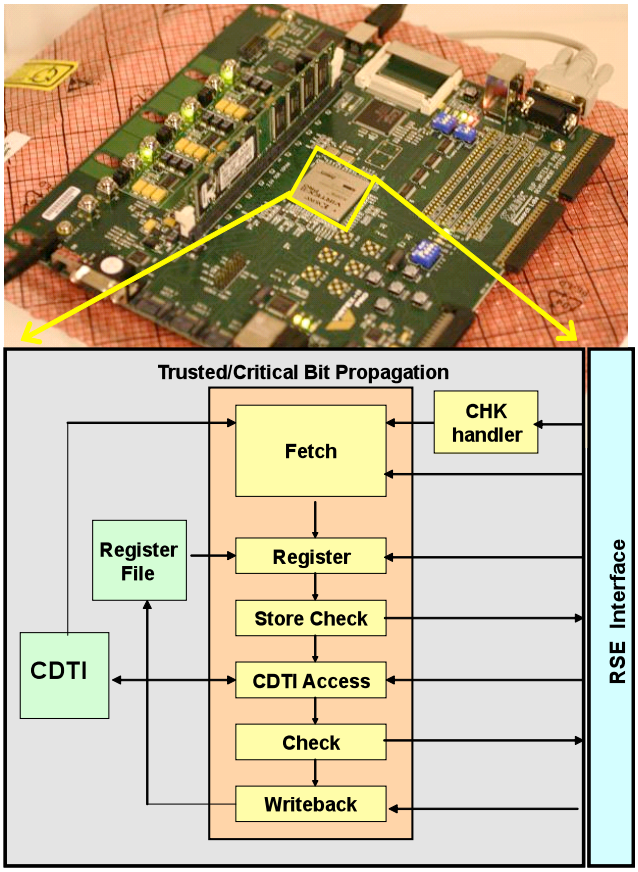

Figure 3: Diagram of the Information Flow Signature Checking Module and Photo of the FPGA Board

add an extra bit to the main bus width within the main processor, or to tag caches with extra information. Such approaches, which have been used in [13] and [14] for different processor-level security enhancements, require significant effort to modify and re-validate the design of the processor and call for changing architectural characteristics, such as bus widths, of current systems.

The current implementation of the CDTI allows for up to 16 4KB pages to be labeled with trusted instructions and trusted or critical data. This corresponds to over 16,000 instructions or memory locations. A 16-entry CAM is used to store the memory pages that are loaded into the CDTI. A 4KB dual-ported RAM is used to store the associated signatures of trusted instructions and trusted or critical data for the loaded pages. The signatures are stored as bit-masks for the memory pages, signifying if a memory location is trusted or critical. The RAM storing the signatures is indexed using tags from the CAM. This structure is similar to a combined instruction and data TLB holding information about working pages.

IFSCM Runtime Operation. The operation of the checking module in Figure 3 is as follows:

- During program initialization: RSE CHK instructions from the main processor pipeline enter the *CHK handler* within the IFSCM and are used to initialize the CDTI.
- During runtime: the *fetch stage* checks if an instruction is trusted within the CDTI, based on the program counter value read from the RSE interface.
- Trusted instructions have their operands are retrieved in the *register stage* of the module.
- The *store check* stage of module enforces Level 1 checking rules for store instructions, before they enter the memory stage of the processor (e.g. if a trusted store instruction uses non-trusted operands, the checking module raises an alarm before the memory operation occurs).
- *CDTI access* looks up and writes back the trusted and critical bit variable information in the CDTI using cache control signals from RSE interface.
- In the *check* stage, trusted instruction operands are checked and the destination of untrusted instructions is checked and actions are taken as shown in Table 1.
- In the *writeback stage*, trusted bit information is propagated back to the IFSCM register file.

IV. Experimental Evaluation

A. Experimental Setup

The RSE and IFCSM are implemented on the Leon 3 soft processor. The system on a chip includes split 16 KB L1 instruction and data caches, an FPU, a DDR memory controller and an ethernet controller. Synthesis of the system is done using Synplify Pro 8.1, and map, place and route are completed using the Xilinx XST 8.2 toolchain. The Xilinx Virtex-II Pro 30 FPGA chip is targeted, with a 65 MHz nominal clock speed for the Leon 3 system.

Our software toolchain is built upon the IMPACT compiler [21]. Its advanced pointer analysis capabilities are ideal for the backwards slicing required by the Information Flow Signatures technique, allowing the technique to be implemented as a compiler pass that requires no programmer intervention. All software performance measurements include initialization times, thus measuring the added overhead of using CHK instructions to initialize the Level 1 checking hardware, and loading the Level 2 checking tables in software.

The applications tested here include the Power and Traveling Salesman (TSP) applications from the Olden benchmark suite [22]. These two programs were determined to have security-critical data which can be impacted by an attacker. The Olden benchmarks have a significant amount of pointer manipulation and have been used to benchmark several previous works. Also, their functionality is more constrained than, for example, HTTP or SSH servers (both of which are analyzed in [15]).

Olden Power. The Olden Power benchmark implements a power-pricing algorithm. Given a set of demands represented by leaves in a tree, it computes the required power output using an iterative optimization problem solver and returns the pricing for the given demands. In this application, we chose the data of the tree holding the set of power demands by the clients as critical. If corrupted, this data could be used to influence prices computed by the application.

Olden TSP. The Traveling Salesman Problem benchmark solves the well-known optimization problem using a partitioning algorithm. The data structure holding the graph used in the problem is a balanced binary tree. We select the pointer to this tree, used throughout the program to access the

Table 2: Compiler Analysis Results for Benchmarks

Benchmark Application	Power	TSP
Total Number of Instructions	10388	5144
Number Trusted Instructions	726 (7.0%)	118 (2.3%)
Number of Trusted and Critical Memory Locations	30	1

tree nodes, as our security-critical data. Malicious tampering with this pointer could lead to an incorrect tree being used for computing the solution.

Table 2 shows the compiler analysis results for the chosen critical variables in Power and TSP.

B. Results

Hardware Area Overhead. Synthesis results for hardware area overheads are displayed Table 3. We find that the largest contribution to the area overhead is the CDTI, at approximately 4.2%. This is not surprising, since the RSE and IFCSM are essentially a set of registers controlled by signals from the main pipeline of the Leon 3 processor, and add less than 0.4% area overhead in this case. The CDTI, however, implements a TLB-like structure, with a 16-entry cam in addition to 4 KB of internal RAM. These area overhead results suggest that on multi-billion transistor chips containing two or more levels of cache hierarchy, the addition of the RSE and IFSCM will have negligible chip area overheads.

Hardware Performance Overhead. Timing constraints were met for the clock frequency of the system defined for the Leon 3. Thus, the RSE and IFSCM have no components that are on the critical timing path. This is because the hardware components used within the IFSCM and CDTI are similar to those used through the rest of the system.

Software Performance Overhead. Since Level 1 checking is run for all instructions, the performance difference between Power and TSP centers on the frequency of the Level 2 checking. In Power, the critical data that we selected was written to sparingly but frequently read from throughout the application. Thus, the Level 2 software checks rarely needed to be called, as they are executed only when trusted data is modified. So, even though the backwards slice of the critical data contained about 7% of the application, it only contributed a 1% performance overhead during execution, as shown in Figure 4.

On the other hand, the backwards slice of the critical data we chose within TSP included less than 2% of the instructions of the application. However, these instructions modified the critical data a substantial number of times during the run of the benchmark. Thus, Level 2 software checks needed to be invoked quite frequently during execution, leading to the 69% performance degradation.

Table 3: Hardware Area Overheads in ASIC gates

Baseline	+ RSE and IFSCM Pipeline	+ CDTI
3.355 million (100%)	3.368 million (100.36%)	3.502 million (104.20%)

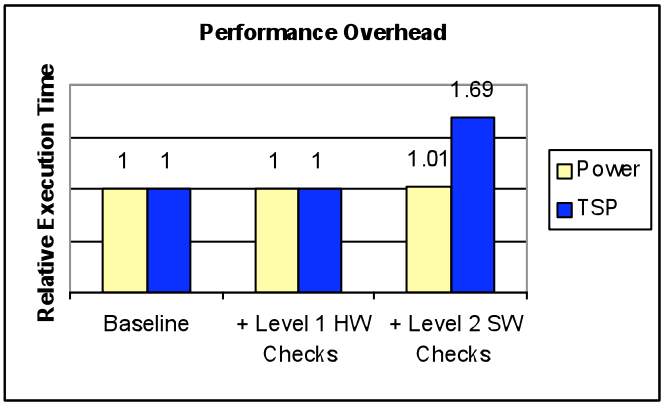

Figure 4: Performance Overheads

V. Discussion and Future Work

A. Looking Forward

Level 2 Checks in Hardware. It is possible to lessen the overhead introduced with Level 2 checks in software by moving them to hardware. Complications arise due to the fact that a trusted memory store instruction may be allowed to write to multiple different objects, as defined by the compiler analysis in Section II.B. However, our experience shows that nearly all instructions are only allowed to write to a single memory object according to application semantics. Thus, by augmenting the CDTI with a hardware lookup table that includes the address range of one object that the trusted instruction is allowed to write to, there is no need to use software Level 2 checks for most of the trusted instructions. The area overhead of such a hardware lookup table will be several times greater than the current CDTI structure.

Extending the CDTI. In order to handle arbitrarily large applications and be able to protect significant portions of an operating system with Information Flow Signatures, the CDTI must be extended to handle an unlimited number of trusted instructions and critical data objects in memory. This can be achieved by piggybacking on the page-handling mechanism already implemented within the MMU. By extending the RSE to receive signals from the TLB that control selection, insertion and removal of TLB entries, the IFSCM could use these signals to control the CDTI. The CDTI will then be synchronized with the set of working memory pages being used by the processor. Pages being removed or inserted into the CDTI can be written and read from main memory using the RSE DMA controller. Assuming a 256-bit memory bus which is typical of today's system architectures, this would add 4 extra memory fetches and 4 memory writes per TLB miss of a dirty page. The Information Flow Signatures stored in main memory could be protected using the IFSCM hardware itself, as a range in main memory could be reserved for this critical data. This will require the IFSCM to have knowledge of the physical addresses being generated by the MMU, in order to check they do not match against the designated area for storing Information Flow Signatures.

IFSCM on SuperScalar Processors. Ironically, the complexity introduced by superscalar processor architecture is helpful to the Information Flow Signatures technique. For example, because of the reorder buffer and store buffers present on these processors, the technique can actually trigger an alarm *before* an instruction or memory operation is committed. Incorporating control signals from these structures into the RSE is straightforward, and has been demonstrated previously on a DLX superscalar processor [19].

Multicore Architectures. Designing the RSE for many-core architectures presents several challenges. In order for the Information Flow Signatures technique to work fully, instructions executed on each core must be checked against a global view of the signatures. Is this best implemented through a single Information Flow Signatures module that has a CDTI containing all working pages from each core? Or, if multiple modules protecting each core separately are used, what is the best method of maintaining coherence between them? These questions raise topics similar to those in Intel processor errata. For example, erratum AH39 for the Core 2 Duo Centrino architecture states "Cache Data Access Request from One Core Hitting a Modified Line in the L1 Data Cache of the Other Core May Cause Unpredictable System Behavior" [2]. Such bugs beg for a more general question: if a component of a processor is faulty, to what extent can we rely on it to provide information to security techniques?

VI. Conclusions

Due to the trend in increasing processor design complexity, greater numbers bugs are introduced in each new architecture. The security implications of processor errata are significant: we suggest that the current paradigm of protecting a system using a virtual fence will not suffice in the near future, as unknown vulnerabilities will be present in the processor or computer system.

In this paper we present the hardware architecture used to enforce the Information Flow Signatures security technique. This combined hardware-software technique allows for trustworthy execution of instructions which influence security-critical data, even in the face of vulnerabilities that exist within a system. The technique detects any deviation from the behavior of the application described by the source code. By using the Reliability and Security Engine as an abstraction to signals of the pipeline, we are able to implement the Information Flow Signatures Checking Module without modification to the processor pipeline. The module itself proves have a small footprint of less than 5% the size of the processor, and has no affect on the performance of the processor. Future extensions to the hardware can lower the performance overhead introduced by the software portion of the technique.

References

[1] B. Bentley, "Validating the Intel(R) Pentium(R) 4 microprocessor", *Proceedings of the Design Automation Conference*, 2001.

[2] Intel Corp., "Intel Core 2 Duo and Intel Core 2 Solo Processor for Intel Centrino Duo Processor Technology: Specification Update", Jan 2008.

[3] "Theo de Raadt Details Intel Core 2 Bugs," http://hardware.slashdot.org/article.pl?sid=07/06/28/1124256. [Accessed Feb 15 2008].

[4] Linus Torvalds, "Core 2 Errata -- problematic or overblown?" http://www.realworldtech.com/forums/index.cfm?action=detail&id=80552&threadid=80534&roomid=2. [Accessed Feb 15 2008.]

[5] "NSF Awards $36 Million Toward Securing Cyberspace", NSF Press Release. http://www.nsf.gov/news/news_summ.jsp?cntn_id=104352 [Accessed Feb. 15, 2008]

[6] C. E. Irvine, K. Levitt, "Trusted Hardware: Can It Be Trustworthy?" *In the Proceedings of Design Automation Conference*, 2007.

[7] Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity," *In Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[8] G. C. Necula, S. McPeak, W. Weimer "CCured: type-safe retrofitting of legacy code," *In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages Portland, Oregon, January 16 - 18, 2002*. POPL '02. ACM Press.

[9] D. Dhurjati, S. Kowshik, V. Adve, "SAFECode: enforcing alias analysis for weakly typed languages," *In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '06). ACM Press, PP. 144-157.

[10] S. Bhatkar, D. DuVarney, and R. Sekar. "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," In V. Paxson, editor, *Proceedings of the 12th USENIX Sec. Symp*., pages 105--20. USENIX, Aug. 2003.

[11] J. Xu, Z. Kalbarczyk, and R. Iyer, "Transparent runtime randomization for security," In A. Fantechi, editor, *Proceeding of the 22nd Symp. on Reliable Distributed Systems*. IEEE Computer Society, Oct. 2003.

[12] E. D. Berger, B. G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," *In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '06). ACM Press, 2006.

[13] M. Dalton, H. Kannan, C. Kozyrakis, "Raksha: A Flexible Information Flow Architecture for Software Security," *in Proc. of the ACM Intl. Symp. on Computer Architecture*, June 9–13, 2007, San Diego.

[14] G. Suh, J. Lee, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," *11th International Conference on Architectural Support for Programming Languages and Operating Systems*. Boston, Massachusetts. October 2004.

[15] W. Healey, et. al. "Ensuring Critical Data Integrity via Information Flow Signatures," University of Illinois Technical Report. UILU-ENG-07-2216. CRHC-07-09.

[16] D. Boneh, R. A. DeMillo, R. J. Lipton, "On the Importance of Eliminating Errors in Cryptographic," *Computations Journal of Cryptology: The Journal of the International Association for Cryptologic Research*, vol. 14, pp. 101-119, 2001.

[17] Adam Boileau. "Hit by a Bus: Physical Access Attacks with Firewire." Presented at Ruxcon 2k6, 2006.

[18] R. Iyer, P. Dabrowski, N. Nakka, Z. Kalbarczyk, "Reconfigurable Tamper resistant Hardware Support Against Insider Threats," *In Proceedings of the 2007 ARO/FSTC Workshop on Insider Attack and Cyber Security*, 2007.

[19] Nakka, N., et. al., "An Architectural Framework for Providing Reliability and Security Support," In *Proceedings of the 2004 international Conference on Dependable Systems and Networks,* 2004.

[20] Jiri Gaisler, Gaisler Research. Leon 3 Synthesizable Processor. http://www.gaisler.com

[21] UIUC Open-IMPACT Effort. The OpenIMPACT IA-64 Compiler. http://gelato.uiuc.edu

[22] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting Dynamic Data Structures on Distributed-Memory Machines," *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, 1995