SIFT: A Low-Overhead Dynamic Information Flow Tracking Architecture for SMT Processors

Meltem Ozsoy, Dmitry Ponomarev and Nael Abu-Ghazaleh *
Department of Computer Science
State University of New York at Binghamton
mozsoy,dima,nael@cs.binghamton.edu

Tameesh Suri †
Intel Corporation
tameesh.suri@intel.com

ABSTRACT

Dynamic Information Flow Tracking (DIFT) is a powerful technique that can protect unmodified binaries from a broad range of vulnerabilities such as buffer overflow and code injection attacks. Software DIFT implementations incur very high performance overhead, while comprehensive hardware implementations add substantial complexity to the microarchitecture, making it unlikely for chip manufacturers to adopt them. In this paper, we propose SIFT (SMT-based DIFT), where a separate thread performing taint propagation and policy checking is executed in a spare context of an SMT processor. However, the instructions for the checking thread are generated in hardware using self-contained off-the-critical path logic at the commit stage of the pipeline. We investigate several optimizations to the base design including: (1) Prefetching of the taint data from shadow memory when the corresponding data is accessed by the primary thread; (2) Optimizing the generation of the taint instructions to remove unneeded instructions. Together, these optimizations reduce the performance penalty of SIFT to 26% on SPEC CPU 2006 benchmarks– much lower than the overhead of previously proposed software-based DIFT schemes. To demonstrate the feasibility of SIFT, we design and synthesize a core with SIFT logic and show that the area overhead of SIFT is only 4.5% and that instruction generation can be performed in one additional cycle at commit time.

Categories and Subject Descriptors

C.1.0 [**Computer Systems Organization**]: Processor Architectures

*This work was done while Nael Abu-Ghazaleh was on a leave of absense at Carnegie Mellon University in Qatar
†This work was done while Tameesh Suri was a PhD student at SUNY Binghamton

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.
CF'11, May 3–5, 2011, Ischia, Italy.
Copyright 2011 ACM 978-1-4503-0698-0/11/05 ...$10.00.

General Terms

Security, Design

Keywords

Security, Microarchitecture, Simultaneous Multithreading, Dynamic Information Flow Tracking

1. INTRODUCTION

Security has emerged as a first-order design consideration in modern computational systems. At the same time, the multi-core and many-core revolution facilitated the emergence of accelerator designs, where separate cores or hardware thread contexts are used to improve the performance and/or reduce the design complexity of various architectural solutions, including those used for security and dynamic fine-grain program monitoring in general. A number of approaches in this domain have similar solution patterns that can be effectively mapped to the accelerators, and several state-of-the-art designs have recently been proposed [15, 7]. In this paper, we investigate a novel architectural technique that utilizes a separate hardware thread context within a modern multithreaded high-performance microprocessor for accelerating and simplifying dynamic information flow tracking.

Dynamic information flow tracking (DIFT) [11, 25, 9, 10, 15, 29, 22, 7, 6, 30] is a technique for defending against a broad range of security exploits, including buffer overflow [3, 17, 8, 4] and other code injection attacks [5, 16]. A DIFT implementation marks (taints) the data entering the system from untrusted sources, and dynamically tracks the propagation of this data during program execution. As a result, it is possible to detect situations where the tainted data is used in potentially insecure ways. For example, when a tainted pointer is dereferenced or a branch target address is tainted, a security exception can be raised. A range of taint checking policies can be implemented to suit the desired protection level for a program or a platform.

Both software [22, 31, 25, 23, 29] and hardware [11, 9, 10, 15, 30, 7] DIFT solutions have been proposed. Software DIFT schemes augment the main program with additional instructions that perform taint propagation and checking. The main drawback of software solutions is the high performance penalty: several-fold slowdown is typical. In addition, software schemes do not address security problems with self-modifying code or multithreaded programs [15].

In response to these limitations, hardware-assisted DIFT

schemes have been proposed [11, 9, 15, 30, 7]. Hardware schemes augment each register and memory location with an additional bit (or several bits if multiple security policies are implemented simultaneously) and perform taint propagation and checking in parallel with the main program execution. Such an approach results in no performance overhead, but requires significant changes to the processor datapath, making it difficult to adapt these designs in practice. The implementations of both hardware and software DIFT schemes have been also demonstrated using multicore processors, but those scheme require the use of either a general purpose [7] or a dedicated [15] co-processor.

The differences among the existing DIFT solutions stem from their approaches to the following two questions: 1) Where in the processor do the DIFT checks take place? and 2) How are these checks implemented? With respect to the first question, the alternatives include performing the checks concurrently with instruction processing using additional hardware resources [11, 9], doing it in separate pre-commit pipeline stages within the same processor [30], using a separate general purpose core [7], or deploying a dedicated co-processor [15]). With respect to the second question, DIFT checks can be performed using dedicated hardware [11, 9], using trace analysis [7] or by executing additional instructions generated through binary translation [29]. Each of these variations has limitations (as discussed in more detail in the related work section).

In this paper, we propose SIFT (SMT-based DIFT) - a novel lightweight DIFT design that uses a separate SMT context for performing DIFT checks in a manner that does not require any software support and mostly utilizes the existing instruction execution infrastructure of a multithreaded core. In particular, SIFT requires no changes to the memory datapath design, does not introduce additional memory structures for storing taint information and does not alter at all the logic within the timing-critical pipeline stages of the core. The key features of SIFT are the following:

- **Separate taint propagation and checking thread within an SMT processor:** First, it performs taint checking and propagation in a separate thread context of an SMT processor, all within a single core. We call this thread the *security thread* in the rest of the paper. Such an approach avoids the use of a separate processing core and associated inter-core communication overhead, but still supports high levels of performance because of concurrent execution of the two threads.
- **Automatically generated security thread instructions:** Second, we propose the hardware-based generation of taint checking and propagation instructions instead of using binary translation. Specifically, the SIFT architecture uses a simple off-the-critical path hardware component at the commit stage of the pipeline, where the committed instructions from the application program (called *primary thread* in the rest of the paper) are dynamically translated into taint checking and propagation instructions to form the security thread. These new instructions are then supplied directly to the fetch queue of the security thread, bypassing the instruction cache.
- **SIFT Optimizations:** Third, we propose several mechanisms to improve the performance of the base SIFT implementation. These include prefetching the data into the security thread's L1 data cache when the primary thread performs respective cache access, and dynamic elimination of checking instructions which are unnecessary for preserving the consistency of taint information. We evaluate the performance impact of SIFT architecture using 23 SPEC 2006 benchmarks and we evaluate the area and timing impact using a combination of synthesis and highly-optimized circuit layouts of the proposed SIFT logic.

The key advantages of the proposed SIFT architecture are the following:

- It requires minimal changes to the processor datapath design. Most of the key datapath components remain unchanged, the additional SIFT logic is implemented as a relatively independent module located off-the-critical path at the commit stage of the pipeline. Our analysis show that the core area increase due to the SIFT logic amounts to 4.5% and at most one additional cycle is required at the commit stage of the pipeline to accommodate the checking instruction generation. For example, this compares favorably with the reported 20% area increase for Raksha design [9] when both the logic and the storage overhead is taking into account. This is a critical advantage for SIFT with respect to other hardware-based schemes, that makes it possible to integrate it with commercial SMT microprocessors.
- It has a modest impact on the application performance. We demonstrate by using 23 SPEC 2006 benchmarks that the basic SIFT scheme results in an average of 43% performance degradation across all benchmarks, and with the proposed performance optimizations the performance loss is reduced to 26%. We note that only simple optimizations were considered for this study (in an effort not to increase the design complexity), additional performance gains would certainly be possible with more elaborate approaches. No other DIFT scheme that fundamentally relies on executing instructions for performing the checks approaches this level of performance, even when implemented using multiple cores [22, 31, 25, 23, 29].
- It faithfully implements the security guarantees afforded by hardware-based DIFT schemes and has identical security properties to them.
- SIFT is transparent to the ISA and the compiler. It does not require any instrumentation of the source code or the binary, and does not require dynamic binary translation. Thus, it can protect unmodified legacy binaries.
- It is easily adaptable to the new threats. All that is needed to adapt the SIFT design to support defense against a new set of security threats is to extend or modify the register maintaining the set of checking policies. This would automatically reconfigure the entire SIFT instruction generation logic. In contrast, adapting hardware-based DIFT schemes requires changes throughout the datapath.
- It can be easily scaled to support a number of different security checking policies simultaneously. In fact, the whole width of the datapath (64-bits in our model) is

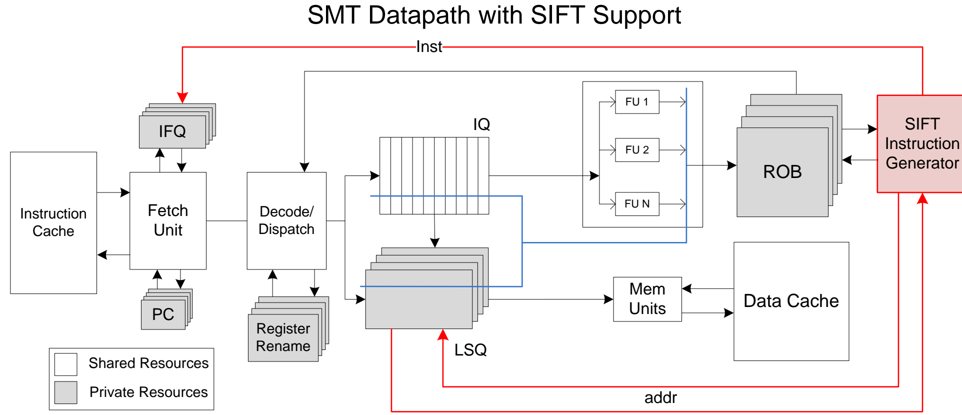

Figure 1: SMT Datapath Augmented with SIFT Logic

available for maintaining the multi-bit taint values for free (in terms of storage and complexity implications compared to the design with single-bit taint values).

The remainder of the paper is organized as follows. Section 2 presents the detailed architecture of SIFT system. Section 3 describes SIFT performance optimizations. Section 4 describes our performance evaluation methodology and presents performance results. In Section 5, we describe the methodology for evaluating the SIFT impact on area and timing and present the results of circuit simulations. Finally, we review the related work in Section 6 and offer our concluding remarks in Section 7.

2. SIFT ARCHITECTURE DETAILS

SIFT implements DIFT by executing a separate security thread in a spare context of a Simultaneously Multithreaded (SMT) processor with hardware-supported generation of checking instructions. SMT processors are mainstream today - for example, the most recent microarchitectures such as Intel's Core I7 [21] and IBM's Power 7 [26] have multithreaded cores. SIFT design allows to utilize one of these thread contexts for security; when such security support is not needed, SIFT logic can be turned off and the context can be used for executing regular applications.

2.1 General SIFT Framework

The proposed SIFT architecture is based on the following two key ideas.

- Execution of the security checking thread in a spare hardware context of an SMT processor in parallel with the primary application thread. The security thread uses general purpose datapath registers and shadow memory locations. We assume, similar to previous DIFT studies, that every memory location is augmented with a shadow copy, which maintains the tainting information about this location. Such multithreaded execution inherently reduces performance losses compared to the execution on a superscalar processor, because the resources of a spare context are used. Furthermore, this arrangement limits the execution of both threads within a boundary of a single core and avoids delays and design complexities associated with inter-thread cross-core communications.
- Hardware-based instruction generation for the security thread. Specifically, when the instructions from the primary thread are processed through the commit stage of the pipeline, they are presented to the security instruction generation logic. This logic produces the checking instructions and supplies them to the security thread, effectively acting as its instruction cache. An important advantage of instruction generation in this manner, compared to software-based DIFT schemes, is that the wrong-path instructions from the primary thread generate no security thread instructions (since they are never committed); this effect reduces the number of instructions executed by the security thread, improving its performance. In addition, as we demonstrate later in the paper, it is also possible to support other optimizations of the generated security thread instructions to further improve performance.

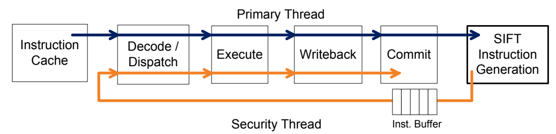

Figure 2: Instruction Flow in SIFT Architecture

The combined effect of these two approaches is that SIFT achieves the positive effects of software-based DIFT schemes (i.e. the low design complexity), but without introducing any changes to the compiler, without requiring binary translation, and keeping the performance overhead of DIFT at modest levels. Therefore, the SIFT architecture effectively combines the best features of both software and hardware schemes. We also note that the primary thread and the security thread synchronize only at the system call boundaries. It has been demonstrated in previous research [15], that such synchronization provides the same security model as instruction-level synchronization. System call monitoring is an established technique for detecting compromised applications [12, 14]. In order to cause real damage, the compromised applications need to use system calls, thus making system call interface a suitable point for detecting violations.

A typical SMT processor datapath augmented with the

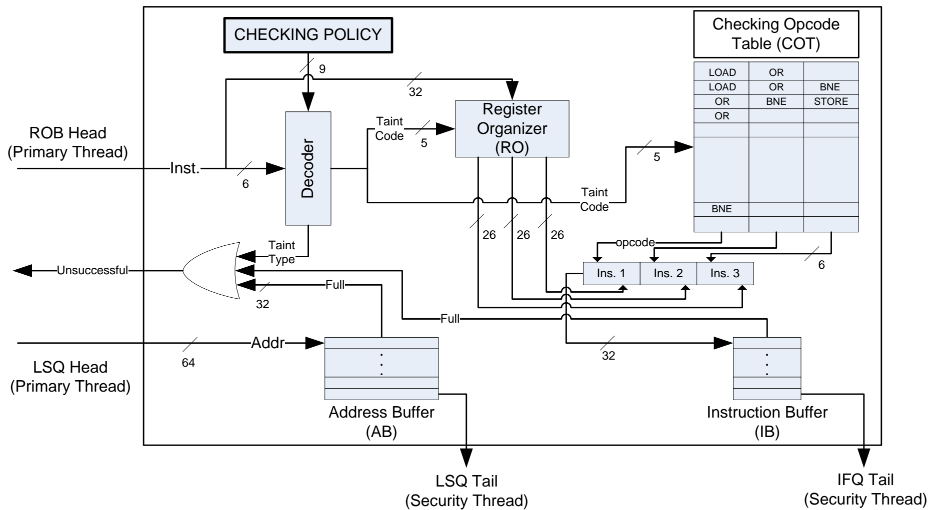

Figure 3: SIFT Instruction Generation Logic

additional logic for implementing SIFT is depicted in Figure 1. Some of the SMT processor resources (such as the register renaming logic, the program counters, and the architectural state) are private to each co-executing thread, and other resources (the out-of-order execution core, the caches) are shared. The SIFT instruction generation logic is shown as a solid box at the commit stage of the pipeline. As seen from this figure, the SIFT logic is localized and it does not impact most of the processor design. We examine the details of the instruction generator later in this section.

Figure 2 depicts the flow of instructions from the primary thread and the security thread through the pipeline. The main aspect to notice is that checking instructions are generated at the commit stage. These instructions enter the pipeline from the instruction buffer, which is fed by the instruction generation logic, bypassing the I-cache and avoiding the I-cache capacity contention between the two threads.

2.2 Taint Checking and Propagation Policies

In this subsection, we describe the rules used for generating taint propagation and checking instructions. These rules are not unique to SIFT, we refer the readers to earlier work [11] where the generated instructions for propagation and policy enforcement are described in more detail; we simply reimplement these for the Alpha instruction set. As such, the security properties of SIFT are identical to those of these prior DIFT schemes.

An attractive feature of SIFT is that the violation detection policies can be flexibly configured through a separate control register, and SIFT logic adjusts to support these variations. This configuration can even be supported during program execution such that each program configures the policies it desires, or even switches policies during execution. In this paper, we evaluate the SIFT framework on the most stringent security policy to gauge the worst-case performance impact; lighter-weight policies will result in less security checking instructions.

As with other DIFT solutions, there are two types of instructions that are generated: taint propagation and violation checking instructions. Propagation instructions are used to track the taint state of data as the program proceeds; these are mostly "OR" instructions for arithmetic operations (ORing the taint values of source registers and associating the resulting taint with the destination register) and "LOAD/STORE" instructions for memory operations. When tainted values are used in unsafe ways, a security exception is raised. The violation detection policies define the rules for when to raise these exceptions. For example, one can prohibit tainted data from being used as a return address or a jump instruction target, preventing code injection attacks [3, 8, 16]. The DIFT system raises an exception if one of these values is tainted according to the Policy Register. The situations that can cause security exceptions are listed below; any subset of these can be enabled, allowing flexible composable security policies. In our simulations, we model the checking overhead when all of these cases are being checked for.

- Tainted address of a load
- Tainted address of a store
- Tainted jump target
- Tainted branch condition
- Tainted system call arguments
- Tainted return address
- Tainted stack pointer
- Tainted memory address AND data
- Tainted memory address OR data

Every checking policy causes a number of additional instructions for implementing the respective checks. Each committed instruction generates between 1 to 3 checking instructions. An exception to this rule is the system call instruction, which requires checking up to 7 arguments for the system that we model. However, since the system calls are infrequent, we do not provide special hardware support to accelerate these checks. The summary of instruction generation is provided in Table 1.

2.3 SIFT Instruction Generation Logic

This subsection presents the Instruction Generation Logic (IGL) - a SIFT component used for generating instructions

Primary Thread	Security Thread
Arithmetic	1 OR Instruction
Memory Instructions	1 OR, 1 Memory, 1 Branch
Branch Instructions	1 Branch
Floating Point Arith	3 Floating Points
FP memory	1 FP Memory, 1 Branch
System Calls	7 Branches

Table 1: Generated Instruction Counts

for the security thread. The internal structure of the IGL block is shown in Figure 3. For simplicity, this figure shows the IGL datapath for processing a single committed instruction.

When an instruction commits, its 6-bit opcode (we assumed Alpha ISA for this study) is used as an index to the opcode decoder. The decoder generates a 5-bit "taint code", which uniquely identifies the sequence of checking instructions corresponding to the instruction being committed. 5 bits are sufficient, because at most 30 combinations of checking instructions can be used in the model that we assumed for this study. This number depends on the checking policies used. A 9-bit checking policy register is another input to the decoder.

The possible combinations of checking instructions are stored in the Checker Opcode Table (COT): one entry for each combination. The generated 5-bit "taint code" is used as an index to the COT which supplies up to three instruction opcodes for the checker thread. Note that the COT only produces the opcodes of the checking instructions, the rest of the instruction bits are derived from Register Organizer (RO) logic, as described below.

The RO block has two inputs: the taint code (generated by the opcode decoder) and the entire 32-bit primary thread instruction that is being committed. The purpose of the RO logic is to either reorganize the remaining 26 bits of the instruction (by switching the positions of some register addresses), or leave them unchanged.

To illustrate the concept of register reorganization, we present two examples. First, consider a shift instruction, where the shift amount is encoded as an immediate value. Register r_1 is the source and register r_{10} is the destination register. The goal of the checking instruction in this case is to propagate the taint value associated with source register r_1 to the destination register r_{10}. This is accomplished by using an OR instruction (BIS instruction in Alpha ISA), with both sources set to r_1 and destination set to r_{10} (see Figure 4). If the source is tainted then the destination (r_{10}) will be tainted after OR operation. Since the original instruction has only one source, but the checking OR instruction has two sources, the address bits corresponding to register r_1 need to be copied to the checking instruction's field containing the address of the second source register. The RO logic performs such register address replication. Also, it inserts the destination register as well as some additional bits to form a complete instruction. In this case (as in the case of other arithmetic instructions), a one-to-one mapping is maintained between the primary thread instructions and the checking instructions.

As a second example, we consider a LOAD instruction. As described in the previous subsection, in the most strict security checking policy, the LOAD instruction sets the taint bit

Figure 4: Register Reorganization: Example 1

of its destination register if either the value being loaded or the address from which it is being loaded are tainted. Furthermore, either of these conditions can result in a security trap, if they are true.

This functionality is implemented via three separate checking instructions, as illustrated in Figure 5. First, the checking load instruction performs the load from the corresponding address in the shadow memory to the destination register. This instruction is analogous to the LOAD instruction from the primary thread. Next, in the second checking instruction, the OR-ing of the taint values associated with the data and the base address are performed in a separate OR (BIS in Alpha) instruction. Note that it is sufficient to simply check the base register, because we assume that the immediate values (values used directly from the instruction field) can not be tainted. The requisite OR instruction needs to have registers arranged in the order "load dest reg, base address register, load dest reg". Again, the RO logic performs the required arrangement of register addresses. Finally, the BNE instruction performs the security check by comparing the resulting taint value of the destination register against zero. The application of this last instruction depends on the checking policy, which is accounted for in the RO logic via the 5-bit "taint code".

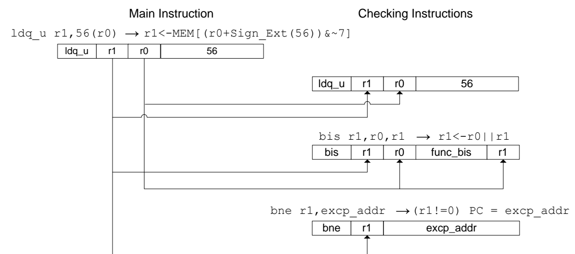

Figure 5: Register Reorganization: Example 2

The detailed schematic of the RO logic is shown in Figure 6. The 5-bit taint code supplied to the RO block controls all of the multiplexers through a control logic. Reorganization of register addresses is performed by a small number of multiplexers and the remaining bits needed to complete the instruction are taken from a small auxilary table. This table contains 10 rows with 11 bits in each entry. This is sufficient, because only a small number of checking instruction types are used. The resulting 26 bits of instruction calculated by the RO logic, along with the 6-bit opcode produced by the COT, form a complete 32-bit long checking instruction which is then stored into the Instruction Buffer (IB) and supplied to the checker thread.

2.4 Communicating Checking Instructions

The checking instruction sequences (at most 3 instructions are generated from every primary thread instruction)

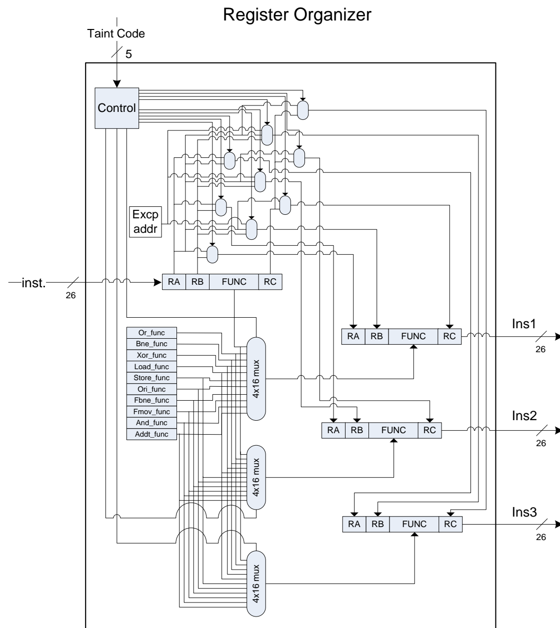

Figure 6: Register Organizer Logic

are stored in the Instruction Buffer (IB) for subsequent consumption by the security thread. Each entry in the IB can be designed to hold three instructions; if less than three are generated, the unused space in the IB remains empty. This arrangement allows the writeback of the entire groups of simultaneously generated checking instructions to be performed through a single write port. A more space-efficient IB design can use separate entries for each instruction, but that requires a larger number of ports to the IB. In our design and simulations, we assumed the latter alternative.

To support momentary spikes in the checking instruction generation rate, a 16-entry IB (with one instruction per entry) is sufficient, as demonstrated by the experiments presented in the results section. When the IB becomes full, the instruction committment process stalls until some space becomes available. Basically, the presence of a free IB entry for the generated checking group is a necessary condition for the commitment of the primary thread instruction that generated this group. Instructions are read from the IB by the checker thread in the order in which they are produced.

2.5 Communicating Memory Addresses

One important issue that needs to be addressed in the proposed design is how to communicate the memory addresses computed by the load and store instructions of the primary thread to the security thread, so that the security thread can perform respective movements of taint information between its registers and shadow memory locations. Indeed, the registers used by the security thread cannot be utilized for anything else other than the storage of taint values. For example, if a load or a store address generated by the primary thread is placed into a register which is used by the security thread, then the taint value previously stored in that register would be destroyed, thus making the security checking process inconsistent.

Our solution to this problem is to directly communicate the effective memory addresses produced by the LOAD and STORE instructions of the primary thread to the security thread through a new structure that we call Address Buffer (AB). The AB is shown on the bottom left of Figure 3. When a LOAD or a STORE instruction commits from the primary thread, its computed effective memory address, which is readily available at this time from the head of the load/store queue (LSQ), is inserted into the tail end of the AB (AB is implemented as a FIFO queue). When the security thread establishes its own LSQ entry for its memory instructions that are being dispatched into the pipeline, it would consult the AB and read the oldest available address into the address field of the just established LSQ entry. At that point, the corresponding entry in the AB can be deallocated. For efficiency, the AB can be placed next to the LSQ, it is shown is part of the IGL box in Figure 3 just for the sake of clarity.

3. PERFORMANCE OPTIMIZATIONS

Although the overhead of SIFT is much lower than that of software based DIFT frameworks, it incurs a performance penalty for the following two reasons: (1) Resource contention: the security thread contends with the primary thread for datapath resources (cache and other datapath structures such as the instruction queue, execution units and the register file); and (2) Load imbalance: the security thread executes several instructions for each primary thread instruction. As a result, the instruction generation buffer fills up causing the primary thread to stall. This section discusses several optimizations that reduce the performance impact of SIFT, while retaining its full information flow tracking capabilities.

3.1 Prefetching for the Security Thread

The first optimization is to use the memory addresses generated by the primary thread to trigger cache prefetching for the security thread. In the SIFT design, the two threads share the same L1 D-cache, with the primary thread performing the accesses to the regular memory locations, and the security thread performing accesses to the shadow memory locations (i.e. locations that keep the tainting information associated with regular memory). Both threads will experience roughly the same number of cache misses, as almost the same set of memory locations will be accessed. Only a small discrepancy in the access patterns occurs due to speculative accesses performed by the wrong-path instructions of the primary thread, which are not replicated by the security thread. In any case, the cache misses result in a significant slowdown for both threads.

The key observation that we exploit in this optimization is that when an instruction in the primary thread makes an access to memory address X, then the security thread will access the same address X in the shadow memory in the near future. Therefore, we propose to prefetch the data from the same location X in shadow memory when the access to this location by the primary thread is encountered. Since a significant slack between the two threads exists (because the checking instructions are generated only after the commitment of the primary thread instructions), timely and highly accurate prefetches are possible. More specifically, the memory addresses generated by all correct-path memory instructions will provide hints for 100% accurate prefetching for the security thread. Wrong-path references could generate security thread prefetches in vain, but those prefetch requests

can be dynamically invalidated once the branch mispredictions are discovered and the primary thread instructions are flushed out of the pipeline.

Specifically, the prefetching mechanism that we propose works as follows. At the time of the primary thread's memory access to address X, we initiate a prefetch to the same address in the shadow memory (we will refer to this address as X_s). When this prefetch request is generated, the L1 cache is first probed for address X_s (using spare cycles when the cache accesses are not performed), and on a miss the memory request for address X_s is sent. Unlike traditional prefetching schemes, in this case the primary thread's address is a direct indicator of the addresses needed by the security thread, and therefore prefetching is always accurate. In addition, this prefetching is also a low-complexity solution, as it does not involve the maintenance of complex prefetching tables with previous execution histories. It is also quite effective in terms of performance improvements, as we demonstrate in the results section.

3.2 Optimizing Generated Instructions

The second optimization targets eliminating redundant taint instructions, or combining instructions [25]. In hardware, it is more difficult to carry out such optimizations dynamically due to the absence of the program structure and the complexity of examining a large window of instructions to detect redundancy. However, simple optimizations are possible in a way that can be easily implemented. Specifically, at the front-end of the instruction generation logic we filter unnecessary instructions according to the following simple rules:

- When a committed instruction from the primary application thread has two source registers and the register addresses of both sources and the register address of the destination are the same, then this instruction does not change the taint value and can be omitted. Indeed, if a corresponding checking instruction was produced, it would be of the form `BIS r1,r1,r1`, which leaves the contents of `r1` unchanged.
- When an instruction has one source register and the register addresses of the source and the destination register are the same, then this instruction can again bypass the IGL. An example of such an instruction is `ADD r1,r1,4`. Whatever the value of register `r1` is before the execution of this instruction, it will remain the same after the execution.

As we demonstrate in the results section, the percentage of checking instructions that can be eliminated with the above rules is substantial. Moreover, each of the rules involves examination of a single instruction and are quite straightforward to support.

Other optimizations that we did not pursue include ISA extensions that combine the common sequences of instructions generated by the taint generation unit and more complex filtering or rewriting of instructions that consider redundancy across multiple generated instructions.

4. PERFORMANCE EVALUATION

In this section, we describe our performance modeling framework and evaluate the performance impact of SIFT architecture.

4.1 Evaluation Methodology

For our studies, we used M-Sim - the execution-driven simulator of an SMT processor, which was developed from Simplescalar [18].

The simulated processor configuration is depicted in Table 2. For our studies we use 23 SPEC CPU2006 [27] benchmarks. The benchmarks were compiled on a native Alpha AXP machine running tru64 unix operating system. Benchmarks were compiled using the native C compiler on DEC Alpha with `-O4 -fast -non_shared` optimization flags. For each benchmark, we simulated 100 million instructions after skipping the initial to 2 Billion instructions to avoid simulating the initialization stages.

Parameter	Configuration
Machine Width	8-wide fetch, issue and commit
Window Size	128-entry ROB, 48-entry LSQ, 32-entry IQ
Physical Registers	256 Integer + 256 FP Physical Registers
L1 I-Cache	64 KB, 2-way set-associative, 64 byte line, 1 cycle hit time
L1 D-Cache	64 KB, 4-way set-associative, 64 byte line, 1 cycle hit time
L2 Unified Cache	512 KB, 16-way set-associative, 64 byte line, 10 cycle hit time
Memory latency	300 cycles

Table 2: Configuration of the Simulated Processor

4.2 Performance Results and Discussions

In this section, we evaluate the performance impact of the proposed architecture and study its performance sensitivity to various parameters. Figure 7-a shows the impact of the IB size on performance of SIFT (without any performance optimizations). On the average across the simulated SPEC 2006 benchmarks, the performance degradation is below 45% and there is little difference with the IB size. For the subsequent experiments, we assumed the IB size of 16 entries. For the individual benchmarks, the performance slowdown ranges from 61% (for *gromacs*) to 26% (for *GemsFDTD*).

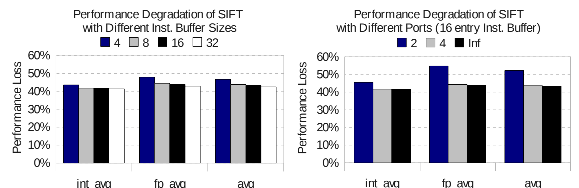

Figure 7: SIFT Overhead a Function of IB Size (a) and Number of IB Ports (b)

Figure 7-b shows performance degradation of SIFT as a function of the number of IB ports. The figure compares the performance of configurations with 2, 4 and infinite number of ports to the IB. As seen from the results, the IB with 4 read and 4 write ports performs almost identical to the IB with the infinite number of ports. Therefore, for the rest of the experiments we assumed a configuration with 4 ports.

Next, we evaluate the impact of the proposed performance optimizations and also present the supporting statistics. First, we examine the elimination (filtering) of the ineffectual instructions. We call this scheme SIFT-F. Figure 8-b shows

the percentage of ineffectual security instructions that are filtered out for each benchmark, and also presents the performance improvements due to such filtering. For individual benchmarks, the percentage of filtered instructions ranges from 17% (for *dealII*) to 45% (for *GemsFDTD*) with the average of about 23%. Consequently, these numbers translate into commensurate performance improvements, although the relative impact of the performance improvement is smaller than the percentage of filtered instructions (simply because the instructions from the primary thread are not being filtered and they represent the constant overhead). On the average across all simulated benchmarks, almost 5% additional performance improvement is observed. In other words, the average performance loss due to SIFT is reduced from 43% before this optimization to about 38% with the optimization. For specific benchmarks, the performance improvement correlates strongly with the percentage of filtered instructions, as demonstrated by Figure 8-a.

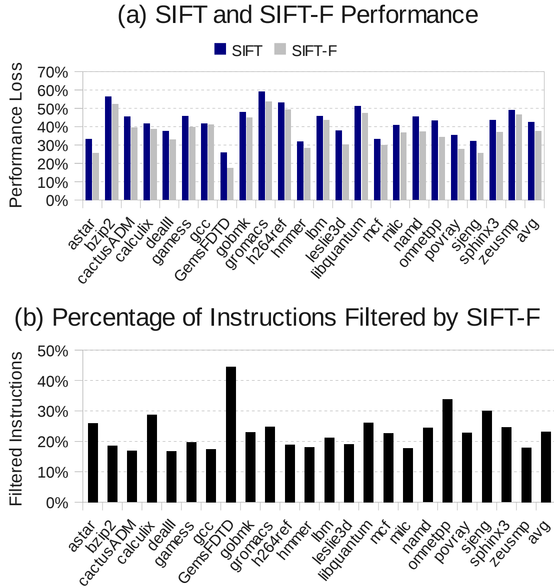

Figure 8: Percentage of Eliminated Checking Instructions and Effect on Performance

Next, we evaluate the impact of our second optimization - security thread's data cache prefetching - on the performance of SIFT. We call this scheme SIFT-P. Figure 9-a shows performance effect of prefetching optimization and also correlates these improvements with the L1 D-cache hit rate experienced by the checker thread in the baseline architecture. As expected, prefetching mechanism has a more significant impact on the programs with higher cache miss rates - this correlation is clearly demonstrated in the figure. On the average across all simulated benchmarks, about 12% reduction in the performance loss due to SIFT is observed.

Finally, Figure 10 summarizes the performance impact of SIFT with all considered optimizations for 8-way processor. In this figure, SIFT-F stands for SIFT with filtered instructions, SIFT-P stands for SIFT with prefetching and and SIFT-FP refers to the case when all performance optimizations are implemented simultaneously. The final performance loss is about 26% for an 8-way machine. Effectively, the effect of the proposed optimizations is synergistic

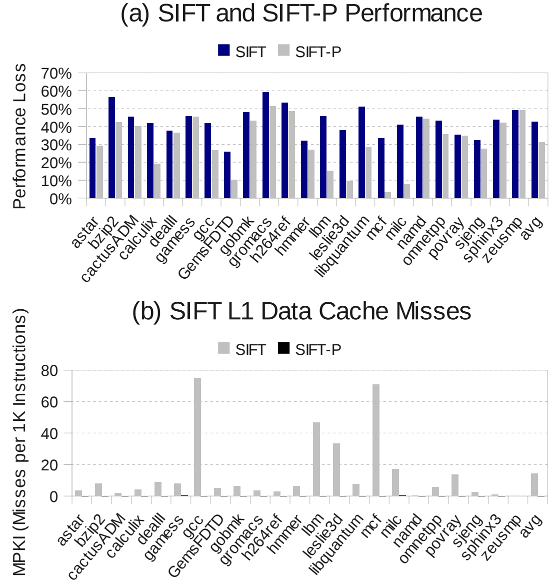

Figure 9: Prefetch Effect on SIFT Performance and SIFT L1 Data Cache MKPI(Misses Per Thousand Instructions)

and they took the performance overhead of SIFT from 43% down to 26%.

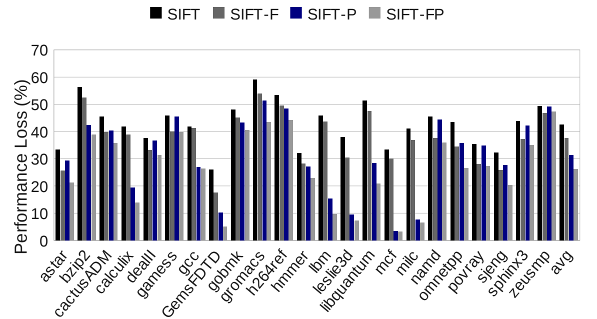

Figure 10: SIFT Performance with All Optimizations

5. AREA AND TIMING EVALUATION

In this section, we describe our methodology for evaluating the area and timing overhead of SIFT logic.

5.1 Circuit Evaluation Methodology

In order to obtain an accurate core area overhead introduced by the SIFT logic, we implemented the instruction generation logic block and integrated it with a single SUN T1 Open source core [1] of an eight-core processor. In this experiment, we scale the parameters of the core to model our baseline architecture, with support for four-way multithreading, as shown in Table 2.

The generation logic was described in RTL where possible, and was synthesized using Synopsys Design Compiler using a TSMC 90nm standard cell library [2]. However, memory structures, such as the COT, IB and AB were implemented

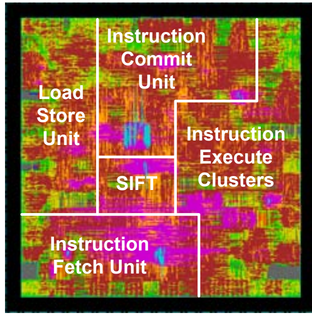

Figure 11: Die Image of the Core

using custom cells designed using industry standard Cadence Virtuoso environment. Custom design for these microarchitectural structures dramatically improves the area efficiency and complexity. Standard synthesis tools implement memory elements as flip-flops, however, for IB and AB, these designs were inefficient because they consume large per-bit area. In addition, the decode logic needed to drive an array of flops require large wire-overhead and strong drivers, which results in further increase in the area overhead. COT was designed to have four read ports, IB with four read and write ports, and AB with a single read and write port.

The integrated processor netlist was placed and routed using Cadence SoC Encounter, to accurately model area and timing overhead due to wire and cell placements. The timing delays of the combinational logic (such as decoders and the RO logic) were accurately modeled using standard cell models provided with TSMC 90nm standard cell library [2]. We used HSPICE circuit simulator from Synopsys to simulate the analog components of the memory structures, such as the sense amps. The wire parasitics (RC delay) were computed using the calculated length of wires from the placed and routed circuit.

5.2 Area Analysis

Figure 11 shows the die image of the modified core with SIFT logic. Our experiments show that the core area increased by about 4.5% with the integrated SIFT unit. We also noted that the IB unit requires about 2% of the area budget. The high area requirement of the IB can be attributed to the large number of ports, which results in a large wire overhead and increased cell area, due to increase in wordline and bitline wires.

Figure 11 shows that the SIFT unit is placed close to the instruction fetch (IF), instruction commit (IC), and the load-store unit (LS), since IB interacts with IF and IC, and AB interacts with LS. It is important to note that since SIFT unit is tightly integrated with multiple modules, it is difficult to distinguish the area overhead of SIFT in Figure 11, which provides a conservative estimate. We note here that SUN T1 processor has in-order execution core, and this is another reason why the area estimate reported here is conservative. The area overhead is likely to be less for an out-of-order core with complex instruction scheduling logic. On SMT processor, this overhead will further reduce because the core supports additional thread contexts. We used Sun T1 architecture, because it was the only open source design

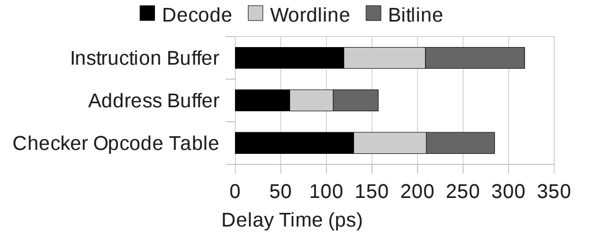

Figure 12: Timing Analysis of COT, AB and IB

available to us. The idea is to obtain the conservative bound on the area estimate.

5.2.1 *Timing Analysis*

The RO and COT are accessed in parallel, and the resulting checking instructions are inserted into the IB. The COT delay is significantly larger than the RO delay in the generation logic, since access to the COT requires decoding and accessing the memory array. Figure 12 shows the timing of three memory arrays used: COT, AB and IB. The timing of SRAM arrays can be classified as $T_{decode}+T_{bitline}+T_{wordline}$ [24]. Figure 12 shows that COT requires more time to decode than IB and AB. COT array is designed to allow access to any of the 32-entries. However, IB and AB are designed as FIFO queues, with a separate read and write counters. Hence, IB and AB decode logic only allows instructions to be written and read from specific contiguous entries, simplifying the decode logic. Figure 12 also shows that IB requires much larger time to drive bitlines and wordlines ($T_{bitline}$, $T_{wordline}$) than other structures. IB requires an extra bitline (per width) and wordline (per entry) to support each additional read port. Write ports are more expensive in terms of area and delay, as they need two times the logic required for a read port.

Overall, the combined access delay of the COT and the IB is about 600 ps for the modelled technology (90nm). The RO access delay is hidden by the COT delay since those two structures are accessed in parallel. Since the SIFT logic operates in parallel with instruction commit, at most one additional cycle is needed to implement SIFT even for a very aggressive implementation with a 3GHz processor (assuming that commit logic operates in a single cycle and the SIFT logic requires 2 cycles). For lower frequency implementations, it may even be possible to implement the entire commit with the new logic within a single cycle. In our performance simulations, we assumed one additional cycle for SIFT processing.

5.3 Complexity and Critical Path Analysis

Previous studies shows that issue logic, including bypass, wakeup and select are decisive in the overall processor clock speed [24]. The SIFT logic (and in particular its IGL unit) interacts with the commit logic, and is not on the critical path of the processor. Synthesis and place and route confirmed that the overall processor cycle time remained unchanged. All that is required to support SIFT is one additional pipeline stage (even for a 3GHz processor) at the commit end of the pipeline. In terms of complexity, the SIFT logic requires a few relatively simple and small structures, which can all be implemented with less than 200 Bytes

of storage and simple combinational circuitry, as detailed in the paper.

6. RELATED WORK

DIFT provides a run time version of earlier works that proposed compile-time static information flow tracking [13, 19, 20]. While the static approach avoids the overhead of runtime information tracking, it is not applicable to a large number of legacy programs written in type-unsafe languages such as C or C++.

Both software [22, 31, 25, 23, 29] and hardware [11, 9, 10, 15, 30, 7] DIFT solutions have been proposed. Software DIFT schemes augment the main program with additional instructions that perform taint propagation and checking. To avoid the need for recompilation, these schemes typically use dynamic binary instrumentation [22, 25]; however, some approaches use source code instrumentation [31]. The main drawback of software solutions is the high performance penalty: several-fold slowdown is typical. This slowdown is a result of executing additional instructions (typically, the number of additional instructions is several times higher than the number of instructions in the original program) and also the overhead of dynamic binary translation.

Source-code based instrumentation [31] has lower performance overhead, but it cannot track information flow in third-party library code and thus will miss security exploits that involve these libraries, as described in US-CERT [28].

Hardware-based DIFT schemes address the performance limitations of software solutions by performing the security checks in hardware, but these schemes require major redesign of the processor datapath. [11, 9, 15, 30, 7]. Such an approach results in no performance overhead, but all key (and often timing-critical) circuitry needs to be augmented with DIFT support. Specifically, all processor registers, cache lines, buses and latches have to be widened to accommodate the taint bits. Such invasive changes complicate the practical adoption of hardware DIFT support by industry. Such designs also have significant area overhead - for example Raksha incurs almost 20% area increase when both logic and storage overhead is taken into account. Moreover, the tight integration of the dift logic with the main datapath retains the overhead even when information flow tracking is not used.

To somewhat address these limitations, FlexiTaint design [30] introduced several new stages prior to the commit stage of the out-of-order processor and accommodated the DIFT checks within these stages, relying on the out-of-order structures to hide the latency. Since FlexiTaint performs DIFT checks prior to the instruction commitment, the main execution and the DIFT operations have to be synchronized at each instruction boundary, as opposed to system call based synchronization, as in our design. Furthermore, FlexiTaint design maintains taint information in separate structures, such as taint register file and taint cache. The additional complexity associated with managing these components is avoided in SIFT, where the taint information is stored and processed in the same way as the regular data. This promotes the simplicity of the SIFT datapath - additional logic is concentrated only at the commit stage for hardware-supported generation of security instructions.

Recently, both hardware and software DIFT schemes have been implemented using multicore processors. To address the design complexity of tightly integrating hardware DIFT within the processor logic, Chen et al [7] proposed the use of a separate core in a multicore chip to perform the DIFT analysis of the execution trace captured by another core (along with other security checks). This approach requires a dedicated core to process the trace (halving the chip's throughput) and also involves additional changes for generating, compressing and decompressing the trace. The hardware overhead, and the resulting performance penalty are both significantly higher than SIFT (slowdown of over 100% in the best case). Another proposed design [15] uses a special-purpose custom designed off-core co-processor for handling DIFT checks. The co-processor synchronizes with the main core on the system call boundaries, capitalizing on the observation that unless a compromised application performs a system call, a potential damage (due to the late detection) is limited. While avoiding the use of a second general-purpose core for DIFT checks, the design of [15] still requires a dedicated co-processor and an interface between the main processor and the co-processor. In [29], a multicore implementation of a software-based DIFT is described, where a new thread is spawned on a helper core to perform DIFT. However, the scheme of [29] still requires a dynamic binary translator to scan the program image and generate a helper thread.

In contrast to the previous hardware DIFT schemes, the SIFT design proposed in this paper only maintains the localized and well-structured hardware for the generation of security checking instructions, while the checking process itself is relegated to the generated software running on an essentially unmodified datapath. As we demonstrate by our experiments, the area overhead of the SIFT logic is less than 5% and it can be accessed within a single cycle at the commit stage of the pipeline. Thus, we believe that the SIFT approach results in important benefits, which are unattainable by either traditional hardware or software schemes.

7. CONCLUDING REMARKS

In this paper, we presented SIFT - a novel architectural implementation of Dynamic Information Flow Tracking. SIFT uses a separate thread to perform taint propagation and policy enforcement. The thread is executed in a spare context of an SMT processor. However, unlike software solutions, the instructions for the taint propagation and policy enforcement thread (security thread) are generated in hardware in the commit stage of the pipeline.

Effectively, SIFT provides run-time taint instruction generation and hardware acceleration for a software-based DIFT framework, but requires no compiler support or support for instrumentation of the source code or the program binary. Compared to software solutions, SIFT incurs lower performance loss because of hardware-accelerated generation of checking instructions and also because of the execution of checking instructions in a separate SMT context. In contrast to architectural DIFT solutions, SIFT preserves the design of all major datapath components. The performance loss of SIFT with all proposed optimizations is only 26% on the SPEC 2006 benchmarks - much lower than the overhead of all earlier proposed software-based solutions.

Finally, we design and synthesize SIFT logic in a representative VLSI process to accurately characterize its area requirement and to verify its impact on the critical path. Conservative estimates show that the area overhead of SIFT is no more than 4.5% and that the checking instruction gen-

eration can be performed within a single additional cycle at the commit stage. Effectively, SIFT support extends the pipeline by a single stage, but has no impact on the instruction throughput or the critical path of the processor. At the same time, SIFT provides exactly the same security guarantees as other hardware-based DIFT schemes.

8. ACKNOWLEDGEMENTS

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-09-1-0137 and by National Science Foundation grants CNS-1018496 and CNS-0958501. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of Air Force Research Laboratory, National Science Foundation, or the U.S. Government.

9. REFERENCES

[1] Opensparc t1 micro architecture specification. In *Sun Microsystems, Inc.*, Mar. 2006.

[2] Tsmc 90nm core library - tcbn90ghp. In *Application Note - Revision 1.2*, Mar. 2006.

[3] Aleph One. Smashing the stack for fun and profit, Nov. 1996.

[4] Cert advisory ca-2001-33: Multiple vulnerabilities in wu-ftpd., Nov. 2001. Available online at `http://www.cert.org/advisories/CA-2001-33.html`.

[5] Cert advisory ca-2002-12: Format string vulnerability in isc dhcpd., May 2002. Available online at `http://www.cert.org/advisories/CA-2002-12.html`.

[6] H. Chen, X. Wu, L. Yuan, B. Z. P. Yew, and F. T.Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. In *ISCA*, June 2008.

[7] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B.Gibbons, T. C.Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA*, June 2008.

[8] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of Usenix Security Symp.*, 1998.

[9] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *ISCA*, June 2007.

[10] M. Dalton, H. Kannan, and C. Kozyrakis. Real world buffer overflow protection for userspace and kernelspace. In *Proc. USENIX Security Symp.*, July 2008.

[11] G. E.Suh, J. W.Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, Oct. 2004.

[12] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, Feb. 2004.

[13] N. Heintze and J. Riecke. The slam calculus: Programming with secrecy and integrity. In *POPL*, 1998.

[14] T. Jim and M. Rajagopalan. System call monitoring using authenticated system calls. In *TDSC*, 2006.

[15] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *DSN*, June 2009.

[16] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM CCS*, 2003.

[17] M. Conover and w00w00 Security Team. w00w00 on heap overflows, Jan. 1999. Available online at `http://www.w00w00.org/files/articles/heaptut.txt`.

[18] M-sim version 3.0, code and documentation, 2005. Available online at: `http://www.cs.binghamton.edu/~msim`.

[19] A. Myers. Jflow: Practical mostly static information flow control. In *POPL*, 1999.

[20] A. Myers and B. Liskov. Proteccting provacy using decentralized lebel model. In *ACM TOSEM, (4), pp.410-422*, 2000.

[21] First the tick, now the tock: Intel microarchitecture (nehalem), 2009. Available online at: `http://www.intel.com/technology/architecture-silicon/next-gen/319724.pdf`.

[22] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, Feb. 2005.

[23] E. Nightingale, D. Peek, and P. Chen. Parallelizing security checks on commodity hardware. In *ASPLOS*, 2008.

[24] S. Palacharla, N. Jouppi, and J. E. Smith. Complexity effective superscalar processors. In *ISCA*, June 1997.

[25] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, Dec. 2006.

[26] B. Sinharoy. Power 7 multicore processor design. In *Keynote talk at MICRO*, Dec. 2009.

[27] C. D. Spradling. Spec cpu2006 benchmark tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, 2007.

[28] Us-cert, 2009. Available online at: `http://www.us-cert.gov/`.

[29] H. K. V. Nagarajan, Y. Wu, and R. Gupta. Dynamic information flow tracking on multicores. In *INTERACT*, Feb. 2008.

[30] G. Venkataramani, I. Doudalis, and Y. Solihin. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *HPCA*, 2008.

[31] W. Xu, E. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. USENIX Security Symp.*, Aug. 2006.