SIFT: Low-Complexity Energy-Efficient Information Flow Tracking on SMT Processors

Meltem Ozsoy*, *Student Member, IEEE,* Dmitry Ponomarev*, *Member, IEEE,*
Nael Abu Ghazaleh*, *Member, IEEE,* Tameesh Suri†, *Member, IEEE,*

Abstract—Dynamic Information Flow Tracking (DIFT) is a powerful technique that can protect unmodified binaries from a broad range of vulnerabilities including buffer overflow and format string attacks. Software DIFT implementations suffer from very high performance overheads, 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. 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 performance 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 code to remove unneeded security instructions; (3) The use of aggregated instructions for collapsing the frequently used groups of security instructions into a single new instruction. Together, these optimizations reduce the performance penalty of SIFT to under 20% on SPEC CPU 2006 benchmarks– much lower than the overhead of previously proposed software-based DIFT schemes. We also analyze the energy overhead of SIFT and show it to be very high - 113% for SPEC 2006 benchmarks. We then propose several techniques that reduce this overhead to only 23%, making SIFT design practical from the energy standpoint. 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.

Index Terms—Security, Microarchitecture, Dynamic Information Flow Tracking, Energy-aware systems.

1 Introduction

DYNAMIC information flow tracking (DIFT) is a security mechanism that marks untrusted data, tracks its propagation, and limits how it can be used [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]; because of its generality, it can defend against a broad range of security exploits, including buffer overflow [13], [14], [15], [16] and other code injection attacks [17], [18]. DIFT works by detecting 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. Several realizations of the general DIFT ideas have been proposed in recent years, including both hardware [1], [3], [4], [5], [10], [8], [12], [11], [19] and software [7], [20], [2], [21], [6], [22] solutions.

Software DIFT implementations inline additional instructions that perform taint propagation and checking with the main program. These extra instructions are inserted either by the compiler, or using binary rewriting. A primary drawback of this approach is the high performance penalty: several-fold slowdown is typical. In addition, software implementations do not address security problems with self-modifying code or multi-threaded programs [5].

**Computer Science Department, Binghamton University, Binghamton, NY 13902–6000. Email: {mozsoy,dima,nael}@cs.binghamton.edu*
†*Intel Corporation, Santa Clara, CA.*

In response to these limitations, hardware-assisted DIFT solutions have been proposed [1], [3], [5], [10], [8]. The idea is to augment each register and memory location with one or more bits to maintain the taint state. Taint propagation and checking is carried out in parallel with the main program execution using dedicated hardware that checks and manipulates the taint state. Such an approach minimizes the performance overhead of DIFT, but requires major changes to the processor datapath, including changes to the timing-critical pipeline stages, and the highly optimized memory hierarchy. As a result, it may be difficult to adopt these designs in practice [5].

In this paper, we propose SIFT (SMT-based DIFT) - a novel, lightweight DIFT design that uses a separate context of a Simultaneously Multithreaded (SMT) processor for performing DIFT checks. SIFT does not require any software support and mostly utilizes existing instruction execution infrastructure of an SMT core to perform security operations. The key idea of SIFT is that taint checking and propagation is performed 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. The instructions for performing security checking are generated in hardware using off-the-critical path logic at the back end of the pipeline. Specifically, the committed instructions from the application program (called *primary thread* in the rest of the paper) are dynamically translated by this logic 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. Figure 1 depicts the flow of instructions from the primary thread and the security thread through the SIFT pipeline. In addition to the baseline SIFT architecture, we also propose a number of optimizations targeted at improving performance and energy-efficiency of SIFT.

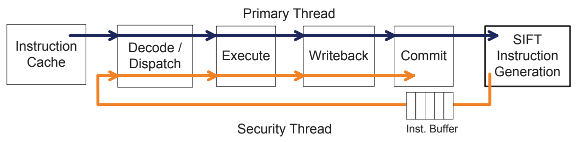

Fig. 1. Instruction Flow in SIFT Architecture

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

- It requires minimal changes to the processor datapath design. 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.
- SIFT logic is concentrated in the commit stage of the pipeline and is located off the critical timing path. This is an important advantage for SIFT compared to most previously proposed hardware-based schemes. This makes it easier to integrate SIFT design with commercial SMT microprocessors.
- SIFT (even with all optimizations) 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.
- SIFT can be easily adapted to new threats and/or 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 available for maintaining the multi-bit taint values for free if so desired.
- It has a modest impact on the application performance and energy consumption when our proposed optimizations are applied.

An important aspect of SIFT architecture is the non-invasive nature of the architectural changes required by the proposed design. In contrast to traditional hardware DIFT solutions, the SIFT logic is contained at the commit stage of the pipeline off the critical timing path. In addition, SIFT does not require any changes to the memory datapath. The low-complexity nature of SIFT changes allows for easier verification and retrofitting of these changes into existing SMT datapath designs. In addition, the performance losses incurred by SIFT are quite modest, although they are higher that those of traditional hardware schemes.

The idea of SIFT was originally presented in our paper published in 2011 ACM Computing Frontiers conference [23]. The current submission extends the original paper in the following ways:

- In addition to evaluating performance, area and cycle time impact of SIFT, this submission also analyzes the implications of SIFT on the processor energy consumption. Our analysis show that the energy overhead of SIFT design, as proposed in [23], is 113% on the average for SPEC 2006 benchmarks. As such a large overhead clearly presents a challenge for the practical adoption of SIFT, this paper proposes several optimizations that reduce the energy overhead of SIFT to about 20%.
- We present a detailed analysis quantifying our design decision to use shared caches between primary and security threads. This design choice avoids the need for implementing separate taint caches or augmenting existing cache lines with taint bits, thus significantly simplifying the memory datapath. We demonstrate that the performance loss due to cache sharing is fairly small: 3% compared to the ideal case where the security thread has perfect memory, but never accesses the primary thread's cache.
- We present a new performance optimization that aggregates multiple security instructions into one instruction, thus further boosting the performance of SIFT.
- The text and the presentation of the CF material have been significantly modified, updated and improved throughout the paper.

2 SIFT DESIGN OVERVIEW

SMT processors are mainstream today - for example, the recent microarchitectures such as Intel's Core I7 [24] and IBM's Power 7 [25] use SMT cores. The SIFT design utilizes 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 Overview of the 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 a SMT processor in parallel with the primary application thread. The security thread uses general purpose datapath registers and shadow memory locations. We assume that every memory location is augmented with a shadow copy, which maintains the taint information about this location.
- 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

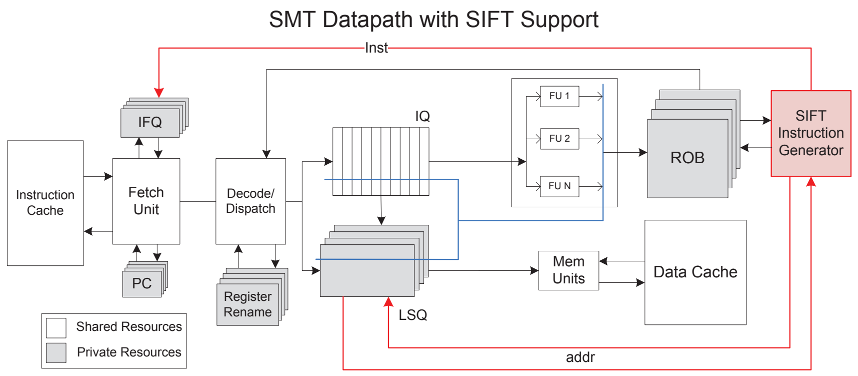

Fig. 2. SMT Datapath Augmented with SIFT Logic

them to the security thread, effectively acting as its instruction cache.

The combined effect of these two approaches is that SIFT achieves the low design complexity due to the reuse of existing datapath, but without introducing any changes to the compiler and without requiring binary translation. All of this is achieved with modest performance, area and energy overhead. In SIFT, the primary thread and the security thread are synchronized only at the system call boundaries. It has been demonstrated in previous research [5], [26], [27], that such synchronization provides the same security model as instruction-level synchronization.

Figure 2 depicts a typical SMT processor datapath augmented with the logic for implementing SIFT. The SIFT instruction generation logic is shown as a solid box at the commit stage of the pipeline.

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 [1] where the generated instructions for propagation and policy enforcement are described in more detail.

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 (OR-ing the taint values of source registers and associating the resulting taint with the destination register) and ”LOAD/STORE” instructions for memory operations. The violation detection policies define the rules for when to raise security exceptions. An attractive feature of SIFT is that the violation detection policies can be flexibly and dynamically configured through a control 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 performance overhead when both data and address of memory operation, system call arguments, conditional branches and jump destinations are checked. However control register can be programmed with a combination of following list of checks.

1. Address of a load, 2. Address of a store, 3. Jump target, 4. Branch condition, 5. System call arguments, 6. Return address, 7. Stack pointer, 8. Memory address AND data, 9. Memory address OR data is tainted

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. The summary of instruction generation is provided in Table 1.

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

2.3 SIFT Instruction Generation Logic

This subsection presents the Instruction Generation Logic (IGL), which is the SIFT component used for generating instructions 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. The possible combinations of checking instructions are stored in the Checker Opcode Table (COT): one entry for each combination. The COT only provides the opcodes of the checking instructions, the rest of the checking instruction bits are derived from

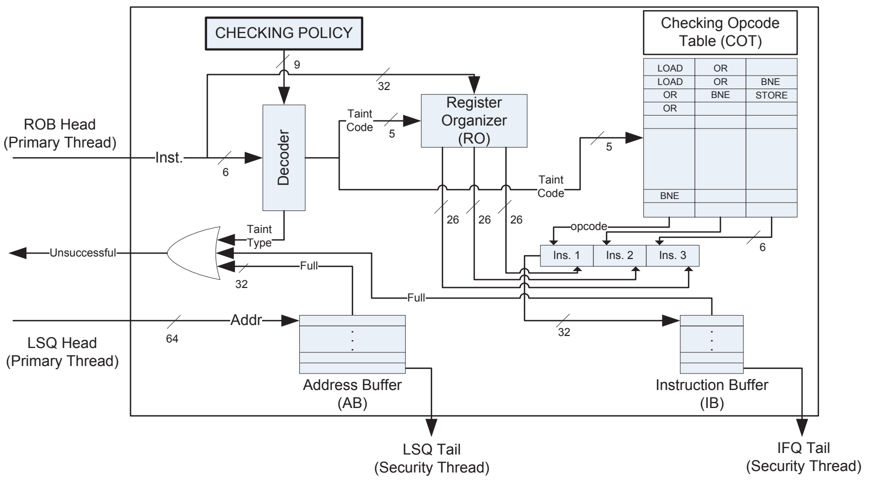

Fig. 3. SIFT Instruction Generation Logic

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

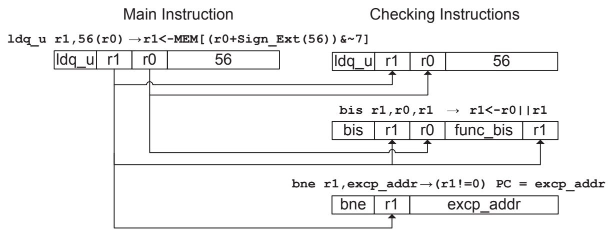
Main Instruction
Checking Instructions
ldq_u r1,56(r0) →r1<-MEM[(r0+Sign_Ext(56))&~7]
ldq_u r1 r0 56
ldq_u r1 r0 56
bis r1,r0,r1 → r1<-r0||r1
bis r1 r0 func_bis r1
bne r1,excp_addr→(r1!=0) PC = excp_addr
bne r1 excp_addr

Fig. 4. Register Reorganization Example

checking instructions, as illustrated in Figure 4. 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. 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 checking instruction sequences are stored in the Instruction Buffer (IB) for subsequent consumption by the security thread. When the IB becomes full, the instruction commitment process stalls until a free entry becomes available. Sensitivty analysis for different buffer sizes is presented in the results section.

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. The difficulty stems from the fact that the registers used by the security thread cannot be utilized for anything else other than the storage of taint values. 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. For efficiency, the AB can be placed next to the LSQ, it is shown as part of the IGL box in Figure 3 just for the sake of clarity.

3 SIFT Performance Optimizations

Although the performance 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 (data cache and other datapath structures such as the instruction queue, execution units and the physical register file); and (2) Load imbalance: the security thread executes several instructions for each primary thread instruction. As a result, the IB 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 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 space in the near future. Therefore, we propose to prefetch the data from the 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.

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. This prefetching scheme is a low-complexity solution, as it does not involve the maintenance of complex prefetching tables based on previous execution histories. It is also quite effective in terms of performance improvements, as we demonstrate in the results section.

3.2 Filtering Security Instructions

The second optimization targets elimination of redundant security instructions. 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 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.

3.3 Aggregating Security Instructions

A third optimization that we pursue is to use additional hardware support for aggregating (fusing) multiple security instructions into one. In particular, for the primary instructions that generate several security instructions as shown in Table 1, we can combine the sequences of these instructions into a single hardware instruction that can be implemented within the existing pipeline. That way, every primary instruction will generate exactly one security instruction. For example, memory instructions can be checked with a new `chk_mem` security instruction which combines OR, memory and branch instructions. When a `chk_mem` instruction proceeds to the dispatch stage of the pipeline, only a single issue queue entry gets allocated to it. After the memory operation is completed, the result is immediately forwarded to the new functional unit, which is composed of one comparator and an OR circuit. Since there is only one instruction in the issue queue and the ROB for the entire aggregated group, this design effectively amplifies the size of these resources, thus increasing performance. Three more new aggregate checking instructions can be defined in this manner, specifically `chk_fp_arith`, `chk_fp_mem` and `chk_syscall` . Since aritmetic operations only generate 1 instruction, they will remain same.

Although these new aggregated security instructions need to be supported in hardware, they are generated internally and do not appear in the application binary. Therefore, this technique does not break the binary compatibility and can still apply to the existing legacy binaries.

4 Power and Energy Analysis and Optimization

SIFT design encounters only modest performance overhead, mainly because of concurrent execution of the two threads. However, a large number of extra instructions executed results in significant power and energy overhead. SIFT generates an average of about 2 security instructions per committed instruction. With filtering, these instructions decrease to 1.5 on average. In this section, we analyze the energy consumption of SIFT and propose two optimizations to reduce it.

Figure 5 shows the energy overhead of SIFT due to the execution of extra instructions. These results were obtained using the Wattch simulator[28]. As seen in the figure, the energy overhead of SIFT is 113% on average across the simulated benchmarks.

To keep explanations clear, and without loss of generality, we describe the optimizations relative to SIFT with a single bit taint; however, both proposed energy and power optimizations generalize to any meta-data size. In order to reduce the SIFT energy overhead, we exploit the fact that SIFT instructions only operate on small size taint values. Therefore, most of the bit slices in the datapath used by the security instructions can be turned-off and/or clock-gated without any impact

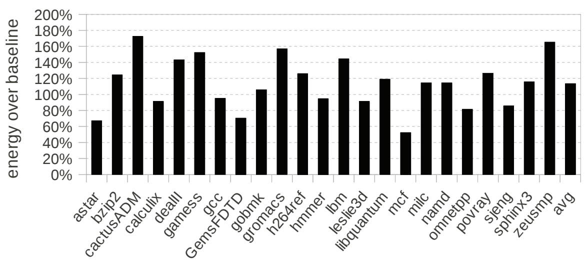

Fig. 5. Energy Overhead of SIFT

on performance. For example, in a 64-bit datapath that we used for our studies, 63 bits can be deactivated, reducing the energy requirements for processing security instructions in SIFT dramatically. In order to accomplish this deactivation, we propose two general approaches that are described below.

4.1 Optimization 1: Bit-Level Clock Gating

The first optimization relies on the fact that the SIFT architecture is amenable to the Deterministic Clock Gating (DCG) scheme proposed in [29]. A functional block that is not in use can be clock-gated (suppressing the clock transition input to it) substantially reducing its power consumption. DCG exploits the advance knowledge of how many instructions are going to be active in each stage of the pipeline to clock-gate the pipeline latches, result buses and ALU slices in a timely fashion without performance overhead. Li et al [29] apply DCG at the instruction granularity. We propose to apply it at the level of individual bits in order to clock-gate all of the bit slices that are unused by security checking instructions. In the case of SIFT, as soon as the security instructions are inserted into the issue queue, we know that they will only need a single bit of the datapath to perform security checks and taint propagation. Of course, the approach generalizes to implementations that maintain multi-bit taint values by enabling the appropriate number of bit-slices.

To distinguish security thread instructions from the main thread instructions, we can rely on the context id bit that is associated with every entry in the issue queue. The context id bits are already used to enable thread-specific instruction squashing on branch mispredictions, so no additional hardware modifications are needed to maintain them. We simply clock gate the inactive bit slices of the security instructions (in the ALU, results buses and pipeline latches), once those instructions are selected for the execution. Figure 6 depicts the clock-gating implementation within the ALU. Here, only a single bit-slice is activated for instructions that execute in the context of the security thread (as determined by the context id).

4.2 Optimization 2: Partitioned On-chip Storage

The second optimization addresses the power and energy consumption within the on-chip storage components such as register files and caches. Since the size of the taint information is known (one-bit), the unused bits of the storage structures hosting them can be turned off. However, in SMTs, the storage components are shared among the different threads, making it difficult to identify which components are used by the taint thread. Moreover, even if we track the components at fine-granularity (say, on a register by register basis), substantial complexity would need to be added to control the fine-grained storage elements.

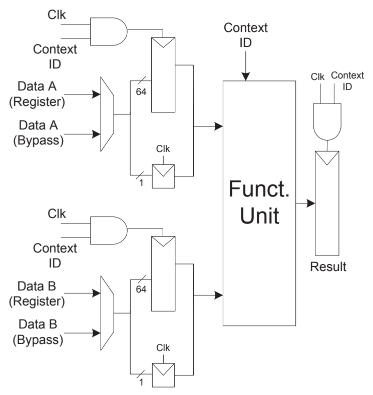

Fig. 6. Clock Gating for Functional Units

To address the above problem, we propose to use a partitioned design. For register files, we propose to partition the physical register file into two halves, such that one half is exclusively used for security instructions and the other half is used by the main thread instructions. The register allocation scheme has to be slightly adjusted to support such partitioning to ensure that only the registers from a predefined partition can be allocated to each thread. Our experiments show that such a small restriction does not have any noticeable impact on the benchmark performance - we observe at most a 1% performance degradation due to this restrictive use of registers.

The benefit of the partitioned design is that we can turn-off all but one bit-slice in the partition that supports the security thread. If the register file is multi-banked (as is the case in modern designs to reduce port requirements) [30], [31], then the partitioned design becomes even more natural, as the power-saving optimizations can be applied to one or several banks supporting security instructions.

We also propose similar optimization for caches (in SIFT, the shared cache is used to hold both security and main thread data). Here, exploiting the fact that for each regular memory access we use one access to the shadow memory (and therefore, the cache capacity is equally split between the main and security instructions), we can statically allocate some cache ways for the main thread and other ways for the security thread. Way selection can be driven by the thread context id. Such partitioned cache design will immediately reduce by half the number of ways that are activated in a set-associate cache for every instruction (as each of the threads only activates their own ways). In addition, the unused bit-slices can be turned-off within the ways allocated for the security thread. Our simulations show that this optimization does not result in any performance degradation due to the fact that the threads have very similar memory access patterns.

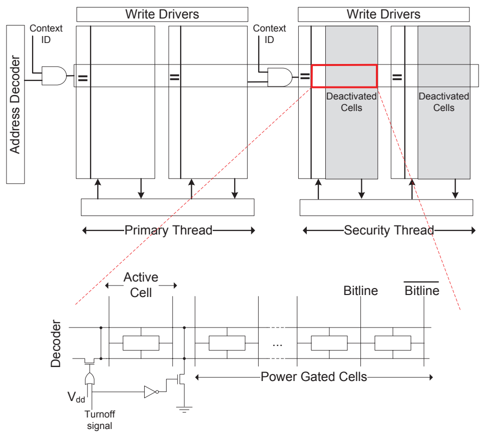

Fig. 7. Partitioned L1 Cache for SIFT-EE

These partitioning optimizations can be activated at the time when the processor switches to SIFT mode. At the time of the switch, the SIFT instruction generation logic will be activated, register file allocation scheme will be altered and the ways of the first-level data cache will be statically partitioned between the main thread and the security thread.

5 SIFT Evaluation Methodology

In order to obtain a conservative estimate of the 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 [32]. The estimate is conservative because SUN T1 core features in-order pipeline. For a processor with out-or-order pipeline, the larger area of the dynamic scheduling engine will reduce the overhead of SIFT.

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 [33]. 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.

For SIFT performance studies, we used M-Sim [34] - an execution-driven simulator of an SMT processor, which was developed in-house from Simplescalar simulator. We used most strict SIFT checking policy described in section 2.2 for all of our simulations unless otherwise stated.

The simulated processor configuration is presented in Table 2.

Parameter	Configuration
Machine Width	8-wide fetch, issue and commit
Window Size	128-entry ROB, 48-entry LSQ, 32-entry IQ
Functional Units and Lat (total/issue)	4 Int Add(1/1), 1 Int Mult(3/1) / Div (20/19), 2 Load/Store (1/1), 4 FP Add (2/1), 1 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
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

We use 23 SPEC CPU2006 [35] benchmarks for this study (we had difficulties compiling the remaining 3 benchmarks for Alpha). 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.

For estimating the energy and power overhead of SIFT, we used Wattch power estimation tool [28] configured for 90nm technology node with structure sizes listed in Table 2.

6 Area and Timing Evaluation

In Figure 8 the die image of the modified core with SIFT logic is depicted. 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 8 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 8, 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

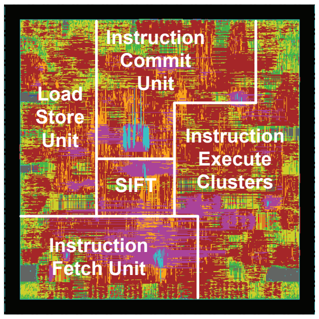

Fig. 8. Die Image of the Core

reported here is conservative since the area of an out-of-order core is significantly larger. 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 available to us to obtain conservative estimates of area and timing. For power evaluation, we use Wattch-based simulations.

We now discuss the timing analysis of the proposed logic. 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 9 shows the timing of three memory arrays used: COT, AB and IB. The timing of SRAM arrays can be classified as $T_{decode} + T_{bitline} + T_{wordline}$ [36]. Figure 9 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 9 also shows that IB requires much larger time to drive bitlines and wordlines ($T_{bitline}$, $T_{wordline}$) than other structures.

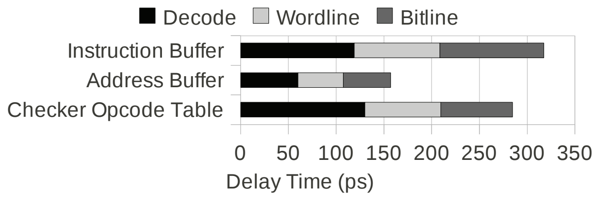

Fig. 9. Timing Analysis of COT, AB and IB

Overall, the combined access delay of the COT and the IB is about 600 ps for the modeled 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 is likely that the entire commit with the new logic can be implemented within a single cycle. In the performance simulations, we assumed one additional cycle for SIFT processing.

7 Evaluation of Energy Optimizations

Figure 10-a shows the energy overhead of SIFT and energy-efficient SIFT (called SIFT-EE in the rest of this section) compared to the baseline processor without SIFT logic activated. Results are shown for each effected processor structure separately. To make the comparisons fair, we assumed that in the baseline execution, only a single 128-register bank of the dual-banked register file is accessed by the application. As seen from the results, the energy reduction achieved in all of these components is significant, with the largest savings encountered within the register file. In fact, the baseline SIFT design almost triples the energy expended in the register file while the optimized SIFT has just a slight overhead since security instructions access only a single bit-slice of the dedicated register file partition. Additionally, Figure 10-b shows the impact of the proposed optimizations on processor power. Total power overhead of SIFT-EE is about 20%. This is lower than the energy overhead, because the program executed longer with SIFT-EE compared to the baseline.

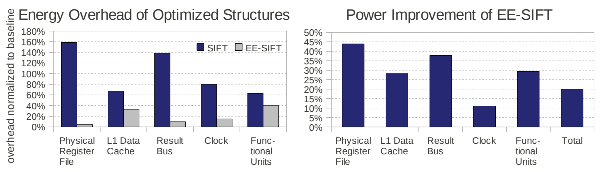

Fig. 10. Energy and Power Overhead of SIFT-EE

In addition to the uneffected components, the extra energy is also expended within the instruction generation unit of SIFT results in the additional 3% energy overhead that accounted for in the results presented in Figure 11.

The energy overhead of SIFT and energy-efficient SIFT on the entire processor is presented in Figure 11. Among the SPEC 2006 benchmarks, *zeusmp* and *cactusADM* suffer the largest energy overhead for the original SIFT - more than 160%. As seen from the graph, the proposed optimizations effectively reduce the energy consumption af all benchmarks, including the one with the highest overhead. On the average across all simulated benchmarks, the energy overhead was reduced to only 23%, which is quite tolerable considering the fact that SIFT fundamentally relies on the execution of additional instructions to provide security.

Note that we presented all our evaluations in terms of pure energy reduction, and did not use unified energy/performance metrics such as energy-delay product. This is because the unified metrics are appropriate for quantifying trade-offs between energy and performance. However, SIFT architecture has overhead in both performance AND energy, therefore it is more straightforward to quantify these overheads separately.

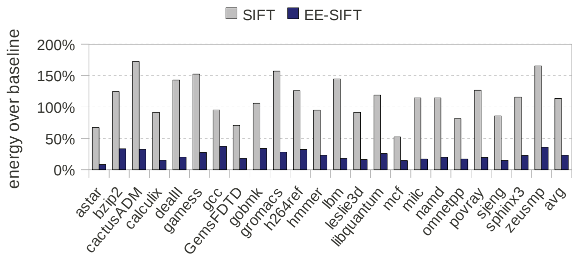

Fig. 11. Energy Overhead of SIFT and SIFT-EE over Baseline

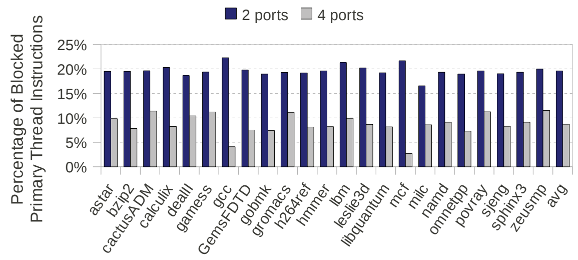

Fig. 13. Percentage of Blocked Main Thread Instructions by Number of IB ports

8 Performance Results & Discussions

In this subsection, we evaluate the performance of the proposed architecture and study its performance sensitivity to various parameters and design decisions.

8.1 SIFT Performance and Sensitivity Analysis

Figure 12-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*).

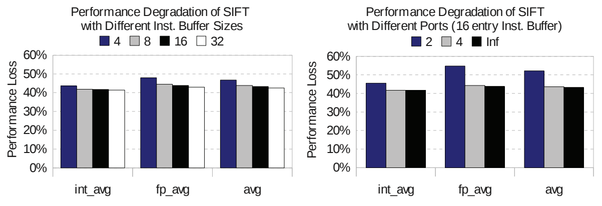

Fig. 12. SIFT Overhead a Function of IB Size (a) and Number of IB Ports (b)

Figure 12-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 remaining experiments we assumed a configuration with 4 ports.

Figure 13 shows the percentage of primary thread instructions that are blocked at the time of commit due to the absense of IB ports. When two IB write ports are used, as many as 20% of all instructions are blocked, while this number drops to 8% for four ports. This explains significant differences in performance between these two points.

8.2 Impact of SIFT Optimizations

First, we consider prefetching optimization and refer to it as SIFT-P in this section. Figure 14-a shows performance effect of SIFT-P and also correlates these improvements with the L1 D-cache hit rate experienced by the security thread in the baseline SIFT architecture. As expected, prefetching mechanism has a more significant impact on applications with higher cache miss rates - this correlation is clearly demonstrated in the Figure 14. *mcf* is one of the benchmarks that benefits most from prefetching optimization. Eventhough it has high miss rates for SIFT beacuse of its data access pattern, it only experiences 33% performance loss which also dropped to around 6% with prefething optimization. On the average across all simulated benchmarks, about 12% reduction in the performance loss due to SIFT is observed.

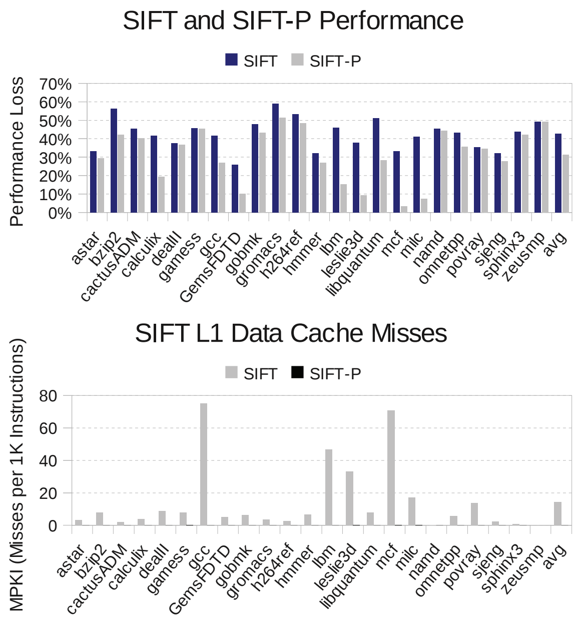

Fig. 14. Prefetching Effect on SIFT Performance and SIFT L1 Data Cache MKPI(Misses Per Kilo Instructions)

The second optimization that we evaluated eliminates (filters) ineffectual and redundant security instructions. We call this scheme SIFT-F. Figure 15-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 the Figure 15-a.

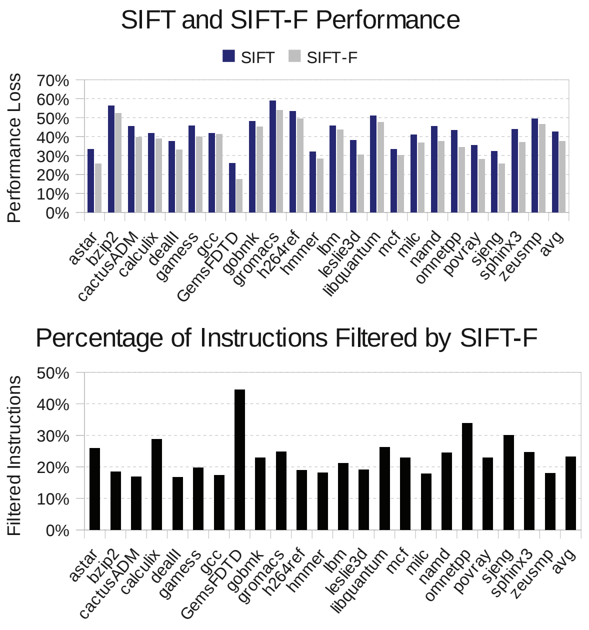

Fig. 15. Percentage of Eliminated Checking Instructions and Effect on Performance

Next, we evaluate the last optimization which provides support for aggregating multiple security instructions into a single instruction. We refer to this variation of SIFT as SIFT-A. Similar to SIFT-F, SIFT-A reduces the number of instructions in the security thread, reducing the pressure on datapath resources. Figure 16 depicts the performance effect of ISA support which decreases the overhead from 43% to 31% on the average. Calculix and leslie3d benchmarks have about 20% performance gain with about 50% fewer checking instructions.

Finally, Figure 17 depicts the performance impact of SIFT with all three optimizations for an 8-way issue processor. In this figure, SIFT-F stands for SIFT with filtered instructions, SIFT-P stands for SIFT with prefetching, SIFT-A stands for SIFT with instruction aggregation, and SIFT-FPA refers to the case when all performance optimizations are implemented simultaneously. The final performance loss is about 20% for an 8-way machine. Effectively, the effect of the proposed optimizations is synergistic and they took the performance overhead of SIFT from 43% down to 20%.

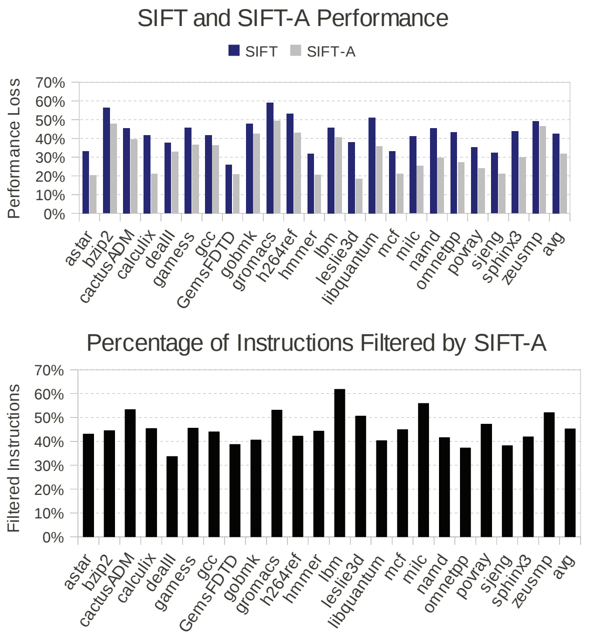

Fig. 16. Impact of SIFT-A on Performance and the Number of Security Instructions

8.3 Performance Impact of SIFT Policy Choice and Processor Width

In this subsection we quantify the effect of checking policy and processor parameters on performance.

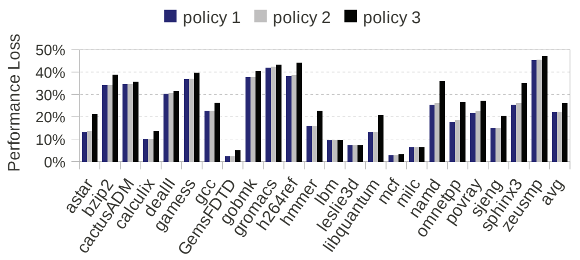

Fig. 18. SIFT-FP Performance with different policies. Policy 1 - taint propagation only, Policy 2 - Jump destinations, stack pointer and return address are checked. Policy 3 - Memory references, system call arguments, conditional branches and jump destinations are checked.

Figure 18 shows SIFT-FP performance for three different policies. While Policy 3 represents the most strict checking policy, Policy 1 does not perform any security checks, but only performs the taint propagation. The interesting result here is that the performance difference between these two policies is only 5%, because the main overhead of SIFT comes from supporting taint propagation.

In order to evaluate SIFT performance with multiple threads, we simulated the scenario where two primary threads are running, but only one of these threads is checked by the SIFT logic. As a metric for comparison,

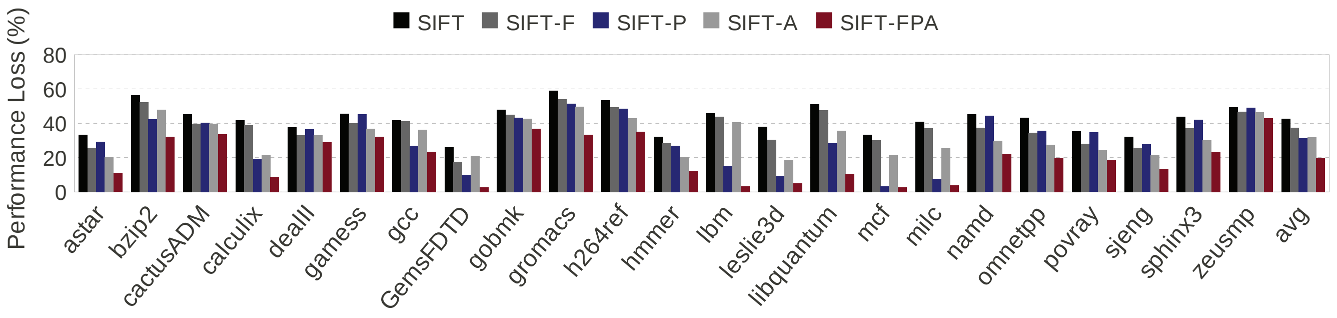

Fig. 17. SIFT Performance with all Optimizations

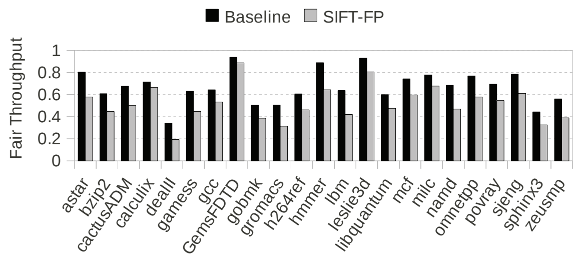

Fig. 19. Comparison of Fair Throughput for SMT and SIFT

we used fairness (or fair throughput) introduced in [37]. As shown in Figure 19, the drop in performance in this new case is just slightly higher than the performance drop experienced by a regular SIFT design with one primary thread and one checker.

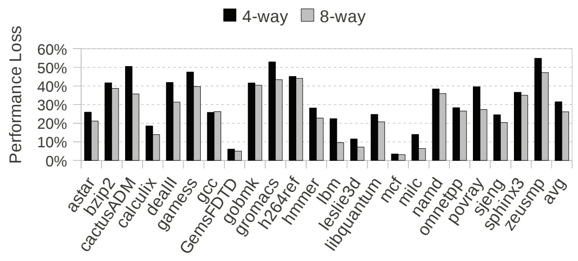

Fig. 20. SIFT-FP Performance with 4-way and 8-way Issue Processors

Finally, we simulated SIFT-FP on a 4-way processor in order to gauge the performance impact on a narrower datapath. Figure 20 shows performance loss of SIFT-FP on 4-way and 8-way processors. While SIFT-FP performance loss 26% on a 8-way machine, it modestly increases to 31% for a 4-way processor.

8.4 Performance Impact of Storing Memory Taint in the L1 Cache

One of the key decisions facing any DIFT design is how to implement the storage for the memory taint values. Some techniques augment the processor caches with the extra bit(s) and store taint information there for each cache line [3]. Other solutions utilize separate taint caches to accelerate memory taint processing [10]. Our solution to this issue was to extend the general design philosophy of SIFT, which is to reuse the existing datapath as much as possible. Therefore, we opted for storing the memory taint information (obtained from the shadow memory space) in the regular L1 and L2 data caches, just like the register taint information is stored in the register file itself. While simple to implement, this design effectively reduces the amount of cache space available for applications, and therefore can lead to performance degradation. In this subsection, we quantify this performance loss and show it to be very small.

Figure 21 shows the effect of cache performance in SIFT architecture. For each benchmark, the figure shows three bars. The leftmost set of bars shows the performance of SIFT-FP design where a 64KB L1 D-cache is dynamically shared between the primary thread and the security thread. The second set of bars shows the performance of a system where the security thread has its own 64KB D-L1 cache, in addition to a 64KB D-L1 cache owned by the primary thread. As seen from the graph, the performance differences are very small - the second design performs just 0.2% better than the first one. Finally, the third set of bars shows the performance with perfect memory system for the security thread. Even this best-case scenario is only less than 4% faster compared to SIFT with cache sharing. These results demonstrate that sharing the cache between the primary and the security threads is an attractive design choice in terms of performance/complexity trade-offs.

We also measured the impact of increased memory pressure on performance. According to our simulaton, the average memory access time increased by 8% with SIFT, which is reflected in our performance results.

Finally, we studied the L1 data cache size effect on performance. Figure 22 depicts the performance loss of SIFT-FP scheme for different L1 data cache sizes. Even with 2KB L1 data cache performance loss for SIFT-FP is around 40%.

In summary, even though the number of executed instructions significantly increases with SIFT, there are several reasons why the performance impact of SIFT is modest. Security instructions are executed in a separate thread context, thus limiting interference with the pri-

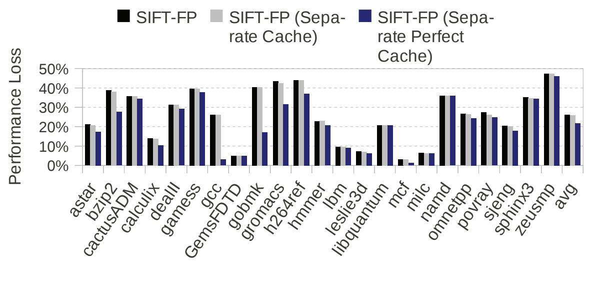

Fig. 21. Shadow Cache Effect on SIFT Performance

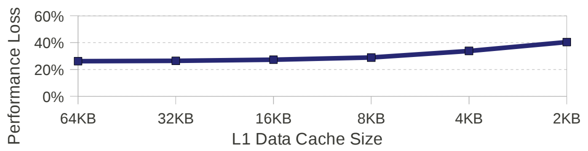

Fig. 22. Impact of the L1 Data Cache Size on SIFT Performance

mary thread instructions. Furthermore, security thread does not have any long-latency instructions and no wrong-path instructions are executed. In terms of extra memory pressure exhibited by SIFT, our experiments showed 8% increase in the average memory access latency with SIFT, this is accounted for in our performance results. In situations where this additional memory pressure results in a serious performancce hit, compression of taint bits can be used to minimize this overhead in systems where it becomes a significant issue - this is left for future work.

9 Related Work

DIFT provides a run time version of earlier works that proposed compile-time static information flow tracking [38], [39], [40]. 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 [7], [20], [2], [21], [6], [22] and hardware [1], [3], [4], [5], [10], [8], [12], [11], [19] DIFT solutions have been proposed. To avoid the need for recompilation, software schemes typically use dynamic binary instrumentation [7], [2], [22]; however, some approaches use source code instrumentation [20]. 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 [20] 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 [41].

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 [1], [3], [5], [10], [8], [12], [11], [19]. Such an approach results in no performance overhead, but all key (and often timing-critical) circuitry needs to be augmented with DIFT support. Such invasive changes complicate the practical adoption of hardware DIFT support by industry. Such designs also have significant area overhead. Moreover, the tight integration of the DIFT logic with the main datapath retains the overhead even when information flow tracking is not used.

In response to these limitations, FlexiTaint [10] 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.

In [42], [12], a new design (called GLIFT) is presented to perform taint checking at gate level with customized circuitry. In addition to explicit flow of information, GLIFT also tracks implicit flows, thus eliminating side channels. In [19], a hybrid monitoring architecture is proposed, where the main core is tightly couples with a reconfigurable fabric. Monitoring functions can be dynamically added to the main core in this design, even after the chip is fabricated.

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 [8] 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 [5] uses a special-purpose custom designed off-core co-processor for handling DIFT checks. While avoiding the use of a second general-purpose core for DIFT checks, the design of [5] still requires a dedicated co-processor and an interface between the main processor and the co-processor. While the performance overhead reported in [5] is minimal, the analysis are only presented for SPEC 2000 integer programs executed on an in-order core. It is unclear if a small dedicated core would be capable of keeping up with the demands of a larger out-of-order superscalar core, especially when workloads with higher ILP levels are executed. In addition, the extensibility of that design is limited, as it is dictated by the hardware constraints of the dedicated core. In contrast, SIFT has the entire 64-bit datapath of the security context that can be utilized with no additional overhead if so desired.

In [6], 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 [6] 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. Table 3 below compares the key features of SIFT with hardware and software DIFT solutions.

The energy reduction techniques proposed in this paper rely on bit-level clock-gating and also the use of partitioned datapath resources such that some bit-slices within these partitions can be deactivated. While clock gating proposed in [29], register file partitioning [43], [44] and cache partitioning [45], [46] have been studied by previous researchers. We show that the design of SIFT has some properties that allow the application of these techniques without performance loss or significant complexity. To the best of our knowledge, no previous DIFT design has been analyzed for energy-efficiency.

	Hardware DIFT	Software DIFT	SIFT
Performance Overhead	0.79% - 3.7%	2x - 37x	20%
Design Complexity	High, Intrusive changes to critical pipeline stages. May impact cycle time	No hardware changes	Modest hardware at the commit stage of the pipeline. Critical path not impacted
Binary Compatibility	Yes	No	Yes
Flexibility	No	High	Supports multi-bit taint for free

TABLE 3
Comparison of SIFT with Hardware and Software DIFT

10 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 20% on the SPEC 2006 benchmarks - much lower than the overhead of all earlier proposed software-based solutions.

We analyzed the energy overhead of SIFT and found it to be 113% across the SPEC 2006 benchmarks. We introduced two optimizations that exploit the small size of the taint meta-data to selectively gate-clock and power-off unneeded portions of the datapath. As a result, the energy overhead of SIFT was reduced to 23%. 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 generation 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.

Acknowledgments

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.

References

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

[2] 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.

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

[4] ——, "Real world buffer overflow protection for userspace and kernelspace," in *Proc. USENIX Security Symp.*, July 2008.

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

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

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

[8] 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.

[9] 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.

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

[11] J. Crandall and F. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, dec. 2004, pp. 221 – 232.

[12] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ASPLOS*, Mar. 2009.

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

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

[15] 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.

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

[17] "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.

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

[19] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh, "Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric," in *MICRO*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 137–148.

[20] 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.

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

[22] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *Computers and Communications, 2006. ISCC '06. Proceedings. 11th IEEE Symposium on*, june 2006, pp. 749 – 754.

[23] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri, "Sift: A low-overhead dynamic information flow tracking architecture for smt processors," in *Proc. of the 8th ACM International Conference on Computing Frontiers (CF'11)*, May 2011.

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

[25] B. Sinharoy, "Power 7 multicore processor design," in *Keynote talk at MICRO*, Dec. 2009.

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

[27] T. Jim and M. Rajagopalan, "System call monitoring using authenticated system calls," in *TDSC*, 2006.

[28] D. Brooks, V. Tivari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *International Symposium on Computer Architecture (ISCA)*, June 2000.

[29] H. Li, S. Bhunia, Y. Chen, T. N. Vijaykumar, and K. Roy, "Deterministic clock gating for microprocessor power reduction," in *In Proc. of 9 th Intl Symp. on High Performance Computer Architecture (HPCA)*, 2003, pp. 113–122.

[30] I. Park, M. D. Powell, and T. N. Vijaykumar, "Reducing register ports for higher speed and lower energy," in *In Proceedings of the International Symposium on Microarchitecture*, 2002, pp. 171–182.

[31] J. H. Tseng and K. Asanović, "Banked multiported register files for high-frequency superscalar microprocessors," in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003.

[32] "Opensparc t1 micro architecture specification," in *Sun Microsystems, Inc.*, Mar. 2006.

[33] "Tsmc 90nm core library - tcbn90ghp," in *Application Note - Revision 1.2*, Mar. 2006.

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

[35] C. D. Spradling, "Spec cpu2006 benchmark tools," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, 2007.

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

[37] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors," in *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, 2001, pp. 164 –171.

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

[39] A. Myers, "Jflow: Practical mostly static information flow control," in *POPL*, 1999.

[40] A. Myers and B. Liskov, "Protecting provacy using decentralized lebel model," in *ACM TOSEM, (4), pp.410-422*, 2000.

[41] "Us-cert," 2009, available online at: http://www.us-cert.gov/.

[42] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security," in *ISCA*. New York, NY, USA: ACM, 2011, pp. 189–200.

[43] S. Wang, J. Hu, S. G. Ziavras, and S. W. Chung, "Exploiting narrow-width values for thermal-aware register file designs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2009.

[44] S. Wang, H. Yang, J. Hu, and S. G. Ziavras, "Asymmetrically banked value-aware register files for low-energy and high-performance," *Microprocess. Microsyst.*, vol. 32, pp. 171–182, May 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1374867.1375319

[45] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," *Computer Architecture, International Symposium on*, vol. 0, p. 0240, 2001.

[46] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *MICRO 34*, 2001.

Meltem Ozsoy is a PhD student in the Department of Computer Science at SUNY Binghamton. Her research interests are in the areas of computer architecture and secure system design.

Dmitry Ponomarev is an Associate Professor in the Department of Computer Science at SUNY Binghamton. His research interests are in the areas of computer architecture, secure system design, power-aware systems and parallel discrete event simulation. He received his PhD from SUNY Binghamton in 2003.

Nael Abu-Ghazaleh is an Associate Professor in the Department of Computer Science at SUNY Binghamton. His research interests are in the areas of secure system design, parallel discrete event simulation, networking and mobile computing. He received his PhD from the University of Cincinnati in 1997.

Tameesh Suri is a performance architect at Intel Corporation, Santa Clara, CA. He received his PhD from SUNY Binghamton in 2009.