Decoupling Dynamic Information Flow Tracking with a Dedicated Coprocessor

Hari Kannan Michael Dalton Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{hkannan, mwdalton, kozyraki}@stanford.edu

Abstract

Dynamic Information Flow Tracking (DIFT) is a promising security technique. With hardware support, DIFT prevents a wide range of attacks on vulnerable software with minimal performance impact. DIFT architectures, however, require significant changes in the processor pipeline that increase design and verification complexity and may affect clock frequency. These complications deter hardware vendors from supporting DIFT.

This paper makes hardware support for DIFT cost-effective by decoupling DIFT functionality onto a simple, separate coprocessor. Decoupling is possible because DIFT operations and regular computation need only synchronize on system calls. The coprocessor is a small hardware engine that performs logical operations and caches 4-bit tags. It introduces no changes to the design or layout of the main processor's logic, pipeline, or caches, and can be combined with various processors. Using a full-system hardware prototype and realistic Linux workloads, we show that the DIFT coprocessor provides the same security guarantees as current DIFT architectures with low runtime overheads.

Keywords: Software security, Semantic Vulnerabilities, Dynamic information flow tracking, Processor architecture, Coprocessors

1 Introduction

Dynamic information flow tracking (DIFT) [10, 19] is a promising technique to detect security attacks on unmodified binaries ranging from buffer overflows to SQL injections [8, 27]. The idea behind DIFT is to tag (taint) untrusted data and track its propagation through the system. DIFT associates a tag with every word of memory. Any new data derived from untrusted data is also tainted using the tag bits. If tainted data is used in a potentially unsafe manner, for instance as a code pointer or as a SQL command, a security exception is immediately raised.

The generality of the DIFT model has led to the development of several implementations. To avoid the need for recompilation [27], most software DIFT systems use dynamic binary translation, which introduces significant overheads ranging from 3x to 37x [18, 20]. Additionally, software DIFT does not work safely with self-modifying and multithreaded programs [6]. Hardware DIFT systems have been proposed to address these challenges [5, 7, 8, 24, 26]. They make DIFT *practical* for all user or library executables, including multithreaded and self-modifying code, and even the operating system itself [9].

Existing DIFT architectures follow two general approaches. *Integrated* architectures provide DIFT support within the main pipeline [5, 7, 8, 24]. While these architectures minimize runtime overhead, they require *significant modifications* to the processor design. All processor registers, pipeline buffers, and internal buses must be widened to accommodate tag bits. Storing tags requires either modification of processor caches, or introduction of an additional tag cache that can be accessed in parallel to the first-level cache. These changes make it difficult for processor vendors to adopt hardware support for DIFT. First, invasive modifications to the processor core increase the design and verification time and may have an impact on the clock frequency. Moreover, the design changes for DIFT are not portable across designs, as DIFT logic is interleaved with conventional logic in a fine-grained manner.

The second architectural approach is to leverage multi-core chips [4]. One core captures a trace of the instructions executed by the application, while another core runs the DIFT analysis on the trace. While this approach offers the flexibility of analysis in software, it introduces significant overheads. It requires a dedicated core to process the trace, which halves the throughput of the overall system, or doubles the power consumption due to the application. The hardware cost is further increased by pipeline changes and custom hardware necessary to produce, compress, and decompress the trace. Compression is necessary to avoid increased contention and power consumption in the multi-core interconnect.

This paper builds upon the FlexiTaint design [26], which implements DIFT similar to the DIVA architecture for reliability checks [2]. It introduces two new stages prior to the commit stage of an out-of-order (OOO) processor pipeline, that accommodate DIFT state and logic. FlexiTaint relies on the OOO structures to hide the latency of the extra stages. By performing DIFT checks and propagation before each instruction commits, FlexiTaint synchronizes regular computation and DIFT on each instruction.

We observe that frequent synchronization performed by the FlexiTaint model is overkill. To maintain the same security model, it is sufficient to synchronize regular computation and DIFT operations at the granularity of system calls. System call monitoring has been established as a well accepted technique

for detecting compromised applications [11, 13]. A compromised application needs to be able to exploit system calls to cause real damage, thus making this interface a great point for detecting errors. Such coarse-grained synchronization allows us to move all DIFT state and logic out of the main core, to a small coprocessor located physically next to the processor core. Our scheme requires no changes to the design or layout of the processor's logic, pipeline, or caches. Hence, it mitigates all risk factors for hardware vendors, as it eliminates the impact of DIFT on the processor's design time, verification time, and clock frequency. Moreover, it allows for portability, as the coprocessor can selectively be paired with multiple processor designs, even in-order cores such as Intel's Atom and Larrabee, and Sun's Niagara.

We describe the coprocessor architecture and its interface to the main core. We also present a prototype that attaches the coprocessor to a SPARC core. By mapping the design to an FPGA board and running Linux, we create a full-featured workstation. We demonstrate that the coprocessor provides the same security features as Raksha, the integrated DIFT architecture that provides comprehensive protection against both memory corruption and high-level security exploits [8, 9]. Specifically, the coprocessor supports multiple and programmable security policies, protects all memory regions (text, heap, stack), correctly handles all types of binaries (dynamically generated, self-modifying, shared libraries, OS, and device drivers), and supports inter-process information flow tracking.

The specific contributions of this work are:

- We describe an architecture that performs all DIFT operations in a small off-core, attached coprocessor. The coprocessor supports a strong DIFT-based security model by synchronizing with the main core only on system calls. No changes are necessary to the main processor's pipeline, design, or layout. The proposed design addresses the complexity, verification time, power, area, and clock frequency challenges of previous proposals for DIFT hardware.
- Using a full-system prototype, we show that the decoupled coprocessor provides the same degree of security as the most complete integrated DIFT architecture. It can protect real-world Linux applications from both low-level and high-level security attacks.
- We show that the coprocessor has a small area footprint (8% of a simple RISC core), and is significantly simpler than log compression and decompression hardware needed for multi-core DIFT. Even with a small cache for tags, the coprocessor introduces less than 1% runtime overhead for the SPECint2000 applications. This is similar to the performance of the integrated DIFT designs and to that of FlexiTaint, despite running stronger security analyses and without an OOO main core. It is also a significant improvement over the multi-core DIFT designs that slow down applications by up to 36%.

Overall, the coprocessor provides a balanced approach for DIFT in terms of performance, cost, complexity, and practicality that is not possible with the known alternatives.

The remainder of the paper is organized as follows. Section 2 provides an overview of hardware DIFT systems. Section 3 presents the design of the DIFT coprocessor, while Section 4 describes the full-system prototype. Section 5 provides an evaluation of the security features, performance, and cost. Finally, Section 6 concludes the paper.

2 Hardware Support for DIFT

Hardware support for DIFT provides a powerful platform for security that can prevent information leaks [22, 25] and protect unmodified binaries against attacks such as buffer overflows and SQL injection [5, 7, 8, 9, 24, 26]. Hardware DIFT systems have been shown to protect against both control and non-control data attacks [9].

2.1 DIFT Background

Hardware DIFT systems logically extend each register and word of memory by a small number of tag bits [5, 7, 8, 24]. The hardware *propagates* and *checks* tags transparently during instruction execution. Tag propagation allows us to track the flow of untrusted data. During instruction execution, hardware propagates the tags of the source operands to the tag of the destination operand. For example, an `add` instruction with a tainted source register will taint the destination register. Tag checks ensure that the instruction does not perform a forbidden operation using a tagged value, such as a data or code pointer dereference. If the check fails, a security exception is raised and control is transferred to a trusted *security monitor*. The monitor may run in kernel mode [5, 7, 24] or user mode [8]. The latter approach makes DIFT applicable to the OS code and allows software analysis to complement hardware checks without high overheads [9].

There is no single propagation or check policy that can prevent all attacks. Raksha, a recent DIFT architecture, provides support for four hardware policies and allows software to manage them through configuration registers [8]. This flexibility is necessary to prevent both high-level and low-level attacks, as well as to adapt to future exploit techniques. Policies are specified at the granularity of *primitive operations* such as move, arithmetic, and logical operations. ISA instructions are decomposed into one or more of these operations before DIFT propagation and checks are applied. This decomposition is necessary to avoid security loopholes that could arise due to the way certain ISAs package multiple operations into a single instruction. Moreover, the decomposition allows propagation and check policies to be independent of the underlying instruction set.

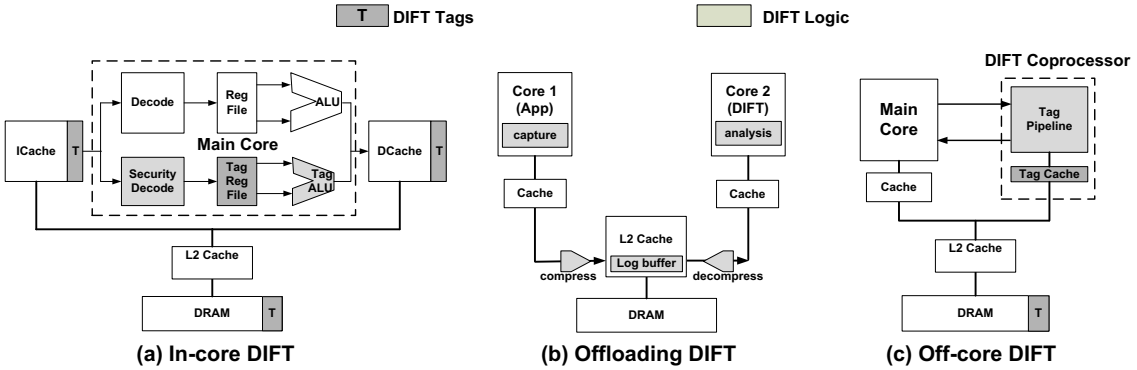

Figure 1: The three design alternatives for DIFT architectures.

2.2 DIFT Design Alternatives

Figure 1 presents the three design alternatives for hardware support for DIFT: (a) the *integrated*, in-core design; (b) the multi-core based, *offloading* design; and (c) an off-core, *co-processor* approach.

Most of the proposed DIFT systems follow the integrated approach, which performs tag propagation and checks in the processor pipeline in parallel with regular instruction execution [5, 7, 8, 24]. This approach does not require an additional core for DIFT functionality and introduces no overhead for inter-core coordination. Overall, its performance impact in terms of clock cycles over native execution is minimal. On the other hand, the integrated approach requires significant modifications to the processor core. All pipeline stages must be modified to buffer the tags associated with pending instructions. The register file and first-level caches must be extended to store the tags for data and instructions. Alternatively, a specialized register file or cache that only stores tags and is accessed in parallel with the regular blocks must be introduced in the processor core. Overall, the changes to the processor core are significant and can have a negative impact on design and verification time. Depending on the constraints, the introduction of DIFT may also affect the clock frequency. The high upfront cost and inability to amortize the design complexity over multiple processor designs can deter hardware vendors from adopting this approach. Feedback from processor vendors has impressed upon us that the extra effort required to change the design and layout of a complex superscalar processor to accommodate DIFT, and re-validate are enough to prevent design teams from adopting DIFT [23].

FlexiTaint [26] uses the approach introduced by the DIVA architecture [2] to push changes for DIFT to the back end of the pipeline. It adds two pipeline stages prior to the final commit stage, which access a separate register file and a separate cache for tags. FlexiTaint simplifies DIFT hardware by requiring few changes to the design of the out-of-order portion of the processor. Nevertheless, the pipeline structure and the processor layout must be modified. To avoid any additional stalls due to accesses to the DIFT tags, FlexiTaint modifies the core to generate prefetch requests for tags early in the pipeline. While it separates regular computation from DIFT processing, it does not fully decouple them. FlexiTaint synchronizes the two on every instruction, as the DIFT operations for each instruction must complete before the instruction commits. Due to the fine-grained synchronization, FlexiTaint requires an OOO core to hide the latency of two extra pipeline stages.

An alternative approach is to offload DIFT functionality to another core in a multi-core chip [3, 4, 16]. The application runs on one core, while a second general-purpose core runs the DIFT analysis on the application trace. The advantage of the offloading approach is that hardware does not need explicit knowledge of DIFT tags or policies. It can also support other types of analyses such as memory profiling and locksets [4]. The core that runs the regular application and the core that runs the DIFT analysis synchronize only on system calls. Nevertheless, the cores must be modified to implement this scheme. The application core is modified to create and compress a trace of the executed instructions. The core must select the events that trigger tracing, pack the proper information (PC, register operands, and memory operands), and compress in hardware. The trace is exchanged using the shared caches (L2 or L3). The security core must decompress the trace using hardware and expose it to software.

The most significant drawback of the multi-core approach is that it requires a full general-purpose core for DIFT analysis. Hence, it halves the number of available cores for other programs and doubles the energy consumption due to the application under analysis. The cost of the modifications to each core is also non-trivial, especially for multi-core chips with simple cores. For instance, the hardware for trace (de)compression uses a 32-Kbyte table for value prediction. The analysis core requires an additional 16-Kbyte SRAM for static information [3]. These systems also require other modifications to the cores, such as additional TLB-like structures to maintain metadata addresses, for efficiency [4]. While the multi-core DIFT approach can also support memory profiling and lockset analyses, the hardware DIFT architectures [8, 9, 26] are capable of performing all the security analyses supported by offloading systems, at a lower cost.

The approach we propose is an intermediate between Flex-

iTaint and the multi-core one. Given the simplicity of DIFT propagation and checks (logical operations on short tags), using a separate general-purpose core is overkill. Instead, we propose to use a small attached coprocessor that implements DIFT functionality for the main processor core and synchronizes with it only on system calls. The coprocessor includes all the hardware necessary for DIFT state (register tags and tag caches), propagation, and checks.

Compared to the multi-core DIFT approach, the coprocessor eliminates the need for a second core for DIFT and does not require changes to the processor and cache hierarchy for trace exchange. Compared to FlexiTaint, the coprocessor eliminates the need for any changes to the design, pipeline, or layout of the main core. Hence, there is no impact on design, verification or clock frequency of the main core. Coarse-grained synchronization enables full decoupling between the main core and the coprocessor. As we show in the following sections, the coprocessor approach provides the same security guarantees and the same performance as FlexiTaint and other integrated DIFT architectures. Unlike FlexiTaint, the coprocessor can also be used with in-order cores, such as Atom and Larrabee in Intel chips, or Niagara in Sun chips.

2.3 Related Work

The proposed DIFT techniques resemble previous architectural approaches to reliability. The DIVA checker architecture [2] verifies the correctness of the executing instruction stream. While both the DIFT coprocessor and DIVA perform a dynamic analysis on the committing instruction stream, they differ in terms of the granularity of synchronization. DIVA has to synchronize the checker and processor on every instruction. The DIFT coprocessor can however, delay synchronization to system calls. This allows us to decouple the DIFT functionality from the main core, giving us the design and verification advantages mentioned earlier. The RSE architecture [17] provides a flexible mechanism to run different reliability and security checks in hardware. This requires heavy integration with the main core, similar to the in-core DIFT designs [5, 8].

The DIFT coprocessor is closest to watchdog processors [15] proposed for reliability checks. A watchdog processor is a simple coprocessor used for concurrent system-level error detection. It monitors the processor's input and output streams, and detects errors pertaining to memory access behavior, control flow, control signals or validity of results. Unlike watchdog processors, the DIFT coprocessor must execute the instructions committed by the main processor, in order to find security flaws.

3 An Off-core Coprocessor for DIFT

The goal of our design is to minimize the cost and complexity of DIFT support by migrating its functionality to a dedicated coprocessor. The main core operates only on data, and has no idea that tags exist. The main core passes information about control flow to the coprocessor. The coprocessor in turn, performs all tag operations and maintains all tag state (configuration registers, register and memory tags). This section describes the design of the DIFT coprocessor and its interface with the main core.

3.1 Security Model

The full decoupling of DIFT functionality from the processor is possible by synchronizing the regular computation and DIFT operations at the granularity of system calls [13, 16, 21]. Synchronization at the system call granularity operates as follows. The main core can commit all instructions other than system calls and traps before it passes them to the coprocessor for DIFT propagation and checks through a coprocessor interface. At a system call or trap, the main core waits for the coprocessor to complete the DIFT operations for the system call and all preceding instructions, before the main core can commit the system call. External interrupts (e.g., time interrupts) are treated similarly by associating them with a pending instruction which becomes equivalent to a trap. When the coprocessor discovers that a DIFT check has failed, it notifies the core about the security attack using an asynchronous exception.

The advantage of this approach is that the main core does not stall for the DIFT core even if the latter is temporarily stalled due to accessing tags from main memory. It essentially eliminates most performance overheads of DIFT processing without requiring OOO execution capabilities in the main core. While there is a small overhead for synchronization at system calls, system calls are not frequent and their overheads are typically in the hundreds or thousands of cycles. Thus, the few tens of cycles needed in the worst case to synchronize the main core and the DIFT coprocessor are not a significant issue.

Synchronizing at system calls implies that a number of additional instructions will be able to commit in the processor behind an instruction that causes a DIFT check to fail in the coprocessor. This, however, is acceptable and does not change the strength of the DIFT security model [13, 16, 21]. While the additional instructions can further corrupt the address space of the application, an attacker cannot affect the rest of the system (other applications, files, or the OS) without a system call or trap to invoke the OS. The state of the affected application will be discarded on a security exception that terminates the application prior to taking a system call trap. Other applications that share read-only data or read-only code are not affected by the termination of the application under attack. Only applications (or threads) that share read-write data or code with the affected application (or thread), and access the corrupted state need to be terminated, as is the case with integrated DIFT architectures. Thus, DIFT systems that synchronize on system calls provide the same security guarantees as DIFT systems that synchronize on every instruction [13].

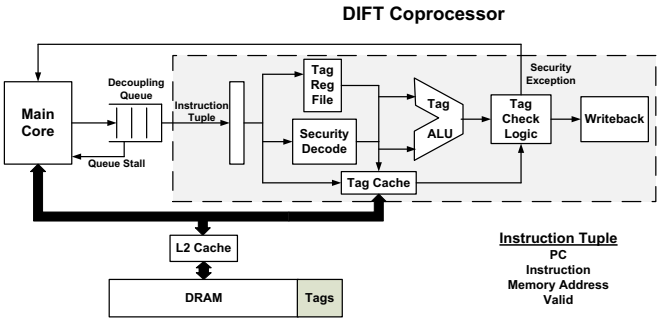

Figure 2: The pipeline diagram for the DIFT coprocessor. Structures are not drawn to scale.

For the program under attack or any other programs that share read-write data with it, DIFT-based techniques do not provide recovery guarantees to begin with. DIFT detects an attack at the time the vulnerability is exploited via an illegal operation, such as dereferencing a tainted pointer. Even with a precise security exception at that point, it is difficult to recover as there is no way to know when the tainted information entered the system, how many pointers, code segments, or data-structures have been affected, or what code must be executed to revert the system back to a safe state. Thus, DIFT does not provide reliable recovery. Consequently, delaying the security exception by a further number of instructions does not weaken the robustness of the system. If DIFT is combined with a checkpointing scheme that allows the system to roll back in time for recovery purposes, we can synchronize the main processor and the DIFT coprocessor every time a checkpoint is initiated.

3.2 Coprocessor Microarchitecture

Figure 2 presents the pipeline of the DIFT coprocessor. Its microarchitecture is quite simple, as it only needs to handle tag propagation and checks. All other instruction execution capabilities are retained by the main core. Similar to Raksha [8], our coprocessor supports up to four concurrent security policies using 4-bit tags per word.

The coprocessor's state includes three components. First, there is a set of configuration registers that specify the propagation and check rules for the four security policies. We discuss these registers further in Section 3.3. Second, there is a register file that maintains the tags for the associated architectural registers in the main processor. Third, the coprocessor uses a cache to buffer the tags for frequently accessed memory addresses (data and instructions).

The coprocessor uses a four-stage pipeline. Given an executed instruction by the main core, the first stage decodes it into primitive operations and determines the propagation and check rules that should be applied based on the active security policies. In parallel, the 4-bit tags for input registers are read from the tag register file. This stage also accesses the tag cache to obtain the 4-bit tag for the instruction word. The second stage implements tag propagation using a tag ALU. This 4-bit ALU is simple and small in area. It supports logical OR, AND, and XOR operations to combine source tags. The second stage will also access the tag cache to retrieve the tag for the memory address specified by load instructions, or to update the tag on store instructions (if the tag of the instruction is zero). The third stage performs tag checks in accordance with the configured security policies. If the check fails (non-zero tag value), a security exception is raised. The final stage does a write-back of the destination register's tag to the tag register file.

The coprocessor's pipeline supports forwarding between dependent instructions to minimize stalls. The main source of stalls are misses in the tag cache. If frequent, such misses will eventually stall the main core and lead to performance degradation, as we discuss in Section 3.3. We should point out, however, that even a small tag cache can provide high coverage. Since we maintain a 4-bit tag per 32-bit word, a tag cache size of T provides the same coverage as an ordinary cache of size $8 \times T$.

3.3 DIFT Coprocessor Interface

The interface between the main core and the DIFT coprocessor is a critical aspect of the architecture. There are four issues to consider: *coprocessor setup*, *instruction flow information*, *decoupling*, and *security exceptions*.

DIFT Coprocessor Setup: To allow software to control the security policies, the coprocessor includes four pairs of registers that control the propagation and check rules for the four tag bits. These policy registers specify the propagation and check modes for each class of *primitive operations*. Their operation and encoding are modeled on the corresponding registers in Raksha [8]. The configuration registers can be manipulated by the main core either as *memory-mapped registers* or as registers accessible through *coprocessor instructions*. In either case, the registers should be accessible only from within a trusted *security monitor*. Our prototype system uses the coprocessor instructions approach. The coprocessor instructions are treated as `nops` in the main processor pipeline. These instructions are used to manipulate tag values, and read and write the coprocessor's tag register file. This functionality is necessary for context switches. Note that coprocessor setup typically happens once per application or context switch.

Instruction Flow Information: The coprocessor needs information from the main core about the committed instructions in order to apply the corresponding DIFT propagation and checks. This information is communicated through a coprocessor interface.

The simplest option is to pass a stream of committed program counters (PCs) and load/store memory addresses from the main core to the coprocessor. The PCs are necessary to identify instruction flow, while the memory addresses are needed because the coprocessor only tracks tags and does not

know the data values of the registers in the main core. In this scenario, the coprocessor must obtain the instruction encoding prior to performing DIFT operations, either by accessing the main core's I-cache or by accessing the L2 cache and potentially caching instructions locally as well. Both options have disadvantages. The former would require the DIFT engine to have a port into the I-cache, creating complexity and clock frequency challenges. The latter increases the power and area overhead of the coprocessor and may also constrain the bandwidth available at the L2 cache. There is also a security problem with this simple interface. In the presence of self-modifying or dynamically generated code, the code in the main core's I-cache could differ from the code in the DIFT engine's I-cache (or the L2 cache) depending on eviction and coherence policies. This inconsistency can compromise the security guarantees of DIFT by allowing an attacker to inject instructions that are not tracked on the DIFT core.

To address these challenges, we propose a coprocessor interface that includes the instruction encoding in addition to the PC and memory address. As instructions become ready to commit in the main core, the interface passes a tuple with the necessary information for DIFT processing (PC, instruction encoding, and memory address). Instruction tuples are passed to the coprocessor in program order. Note that the information in the tuple is available in the re-order buffer of OOO cores or the last pipeline register of in-order cores to facilitate exception reporting. The processor modifications are thus restricted to the interface required to communicate this information to the coprocessor. This interface is similar to the lightweight profiling and monitoring extensions recently proposed by processor vendors for performance tracking purposes [1]. The instruction encoding passed to the coprocessor may be the original one used at the ISA level or a predecoded form available in the main processor. For x86 processors, one can also design an interface that communicates information between the processor and the coprocessor at the granularity of micro-ops. This approach eliminates the need for x86 decoding logic in the coprocessor.

Decoupling: The physical implementation of the interface also includes a *stall* signal that indicates the coprocessor's inability to accept any further instructions. This is likely to happen if the coprocessor is experiencing a large number of misses in the tag cache. Since the locality of tag accesses is usually greater than the locality of data accesses (see Section 3.4), the main core will likely be experiencing misses in its data accesses at the same time. Hence, the coprocessor will rarely be a major performance bottleneck for the main core. Since the processor and the coprocessor must only synchronize on system calls, an extra queue can be used between the two in order to buffer instruction tuples. The queue can be sized to account for temporary mismatches in processing rates between the processor and the coprocessor. The processor stalls only when the decoupling queue is full or when a system call instruction is executed.

To avoid frequent stalls due to a full queue, the coprocessor must achieve an instruction processing rate equal to, or greater than, that of the main core. Since the coprocessor has a very shallow pipeline, handles only committed instructions from the main core, and does not have to deal with mispredicted instructions or instruction cache misses, a single-issue coprocessor is sufficient for most superscalar processors that achieve IPCs close to one. For wide-issue superscalar processors that routinely achieve IPCs higher than one, a wide-issue coprocessor pipeline would be necessary. Since the coprocessor contains 4-bit registers and 4-bit ALUs and does not include branch prediction logic, a wide-issue coprocessor pipeline would not be particularly expensive. In Section 5.2.2, we provide an estimate of the IPC attainable by a single-issue coprocessor, by showing the performance of the coprocessor when paired with higher IPC main cores.

Security Exceptions: As the coprocessor applies tag checks using the instruction tuples, certain checks may fail, indicating potential security threats. On a tag check failure, the coprocessor interrupts the main core in an asynchronous manner. To make DIFT checks applicable to the operating system code as well, the interrupt should switch the core to the trusted security monitor which runs in either a special trusted mode [8, 9], or in the hypervisor mode in systems with hardware support for virtualization [12]. This allows us to catch bugs in both userspace and in the kernel [9]. The security monitor uses the protection mechanisms available in these modes to protect its code and data from a compromised operating system. Once invoked, the monitor can initiate the termination of the application or guest OS under attack. We protect the security monitor itself using a sandboxing policy on one of the tag bits. For an in-depth discussion of exception handling and security monitors, we refer the reader to related work [8]. Note, however, that the proposed system differs from integrated DIFT architectures only in the synchronization between the main core and the coprocessor. Security checks and the consequent exception processing (if necessary) have the same semantics and operation in the coprocessor-based and the integrated designs.

3.4 Tag Cache

The main core passes the memory addresses for load/store instructions to the coprocessor. Since the instruction is communicated to the coprocessor after the main processor completes execution, the address passed can be a physical one. Hence, the coprocessor does not need a separate TLB. Consequently, the tag cache is physically indexed and tagged, and does not need to be flushed on page table updates and context switches.

To detect code injection attacks, the DIFT core must also check the tag associated with the instruction's memory location. As a result, tag checks for load and store instructions require two accesses to the tag cache. This problem can be eliminated by providing separate instruction and data tag caches,

similar to the separate instruction and data caches in the main core. A cheaper alternative that performs equally well is using a unified tag cache with an L0 buffer for instruction tag accesses. The L0 buffer can store a cache line. Since tags are narrow (4 bits), a 32-byte tag cache line can pack tags for 64 memory words providing good spatial locality. We access the L0 and the tag cache in parallel. For non memory instructions, we access both components with the same address (the instruction's PC). For loads and stores, we access the L0 with the PC and the unified tag cache with the address for the memory tags. This design causes a pipeline stall only when the L0 cache misses on an instruction tag access, and the instruction is a load or a store that occupies the port of the tag cache. This combination of events is rare.

3.5 Coprocessor for In-Order Cores

There is no particular change in terms of functionality in the design of the coprocessor or the coprocessor interface if the main core is in-order or out-of-order. Since the two synchronize on system calls, the only requirement for the main processor is that it must stall if the decoupling queue is full or a system call is encountered. Using the DIFT coprocessor with a different main core may display different performance issues. For example, we may need to re-size the decoupling queue to hide temporary mismatches in performance between the two. Our full-system prototype (see Section 4) makes use of an in-order main core.

3.6 Multiprocessor Consistency Issues

For multiprocessors where each core has a dedicated DIFT coprocessor, there are potential consistency issues due to the lack of atomicity of updates for data and tags. The same problem occurs in multi-core DIFT systems and FlexiTaint, since metadata propagation and checks occur after the processing of the corresponding instruction completes.

The atomicity problem is easy to address in architectures with weak consistency models by synchronizing the main core and the DIFT coprocessor on memory fences, acquires, and releases in addition to system calls and traps. This approach guarantees that tag check and propagation for all instructions prior to the memory fence are completed by the coprocessor before the fence instructions commit in the processor. The atomicity problem is harder to solve in systems with stricter consistency models such as sequential consistency. One approach is to use transactional memory as detailed in [6]. A dynamic binary translator (DBT) is used to instrument the code with transactions that encapsulate the data and metadata accesses for one or more basic blocks in the application. A more complex approach is to actually provide atomicity of individual loads and stores in hardware. This entails keeping track of coherence requests issued by different cores in the system in order to detect when another access is ordered in between the data and metadata accesses for an instruction.

Parameter	Specification
Leon pipeline depth	7 stages
Leon instruction cache	8 KB, 2-way set-associative
Leon data cache	16 KB, 2-way set-associative
Leon instruction TLB	8 entries, fully associative
Leon data TLB	8 entries, fully associative
Coprocessor pipeline depth	4 stages
Coprocessor tag cache	512 Bytes, 2-way set-associative
Decoupling queue size	6 entries

Table 1: The prototype system specification.

4 Prototype System

To evaluate the coprocessor-based approach for DIFT, we developed a full-system FPGA prototype based on the SPARC architecture and the Linux operating system. Our prototype is based on the framework for the Raksha integrated DIFT architecture [8]. This allows us to make direct performance and complexity comparisons between the integrated and coprocessor-based approaches for DIFT hardware.

4.1 System Architecture

The main core in our prototype is the Leon SPARC V8 processor, a 32-bit synthesizable core [14]. Leon uses a single-issue, in-order, 7-stage pipeline that does not perform speculative execution. Leon supports SPARC coprocessor instructions, which we use to configure the DIFT coprocessor and provide security exception information. We introduced a decoupling queue that buffers information passed from the main core to the DIFT coprocessor. If the queue fills up, the main core is stalled until the coprocessor makes forward progress. As the main core commits instructions before the DIFT core, security exceptions are imprecise.

The DIFT coprocessor follows the description in Section 3. It uses a single-issue, 4-stage pipeline for tag propagation and checks. Similar to Raksha, we support four security policies, each controlling one of the four tag bits. The tag cache is a 512-byte, 2-way set-associative cache with 32-byte cache lines. Since we use 4-bit tags per word, the cache can effectively store the tags for 4 Kbytes of data.

Our prototype provides a full-fledged Linux workstation environment. We use Gentoo Linux 2.6.20 as our kernel and run unmodified SPARC binaries for enterprise applications such as Apache, PostgreSQL, and OpenSSH. We have modified a small portion of the Linux kernel to provide support for our DIFT hardware [8, 9]. The security monitor is implemented as a shared library preloaded by the dynamic linker with each application.

4.2 Design Statistics

We synthesized our hardware (main core, DIFT coprocessor, and memory system) onto a Xilinx XUP board with an XC2VP30 FPGA. Table 1 presents the default parameters for the prototype. Table 2 provides the basic design statistics for our coprocessor-based design. We quantify the additional

Component	BRAMs	4-input LUTs
Base Leon core (integer)	46	13,858
FPU control & datapath Leon	4	14,000
Core changes for Raksha	4	1,352
% Raksha increase over Leon	**8%**	**4.85%**
Core changes for coprocessor IF	0	22
Decoupling queue	3	26
DIFT coprocessor	5	2,105
Total DIFT coprocessor	8	2,131
% coprocessor increase over Leon	**16%**	**7.64%**

Table 2: Complexity of the prototype FPGA implementation of the DIFT coprocessor in terms of FPGA block RAMs and 4-input LUTs.

resources necessary in terms of 4-input LUTs (lookup tables for logic) and block RAMs, for the changes to the core for the coprocessor interface, DIFT coprocessor (including the tag cache), and the decoupling queue. For comparison purposes, we also provide the additional hardware resources necessary for the Raksha integrated DIFT architecture. Note that the same coprocessor can be used with a range of other main processors: processors with larger caches, speculative execution etc. In these cases, the overhead of the coprocessor as a percentage of the main processor would be even lower in terms of both logic and memory resources.

5 Evaluation

This section evaluates the security capabilities and performance overheads of the DIFT coprocessor.

5.1 Security Evaluation

To evaluate the security capabilities of our design, we attempted a wide range of attacks on real-world applications, and even one in kernelspace using unmodified SPARC binaries. We configured the coprocessor to implement the same DIFT policies (check and propagate rules) used for the security evaluation of the Raksha design [8, 9]. For the low-level memory corruption attacks such as buffer overflows, hardware performs taint propagation and checks for use of tainted values as instruction pointers, data pointers, or instructions. Synchronization between the main core and the coprocessor happens on system calls, to ensure that any pending security exceptions are taken. For high-level semantic attacks such as directory traversals, the hardware performs taint propagation, while the software monitor performs security checks for tainted commands on sensitive function and system call boundaries similar to Raksha [8]. For protecting against Web vulnerabilities like cross-site scripting, we apply this tainting policy to Apache, and any associated modules like PHP.

Table 3 summarizes our security experiments. The applications were written in multiple programming languages and represent workloads ranging from common utilities (gzip, tar, polymorph, sendmail, sus) to server and web systems (scry, htdig, wu-ftpd) to kernel code (quotactl). All experiments were performed on unmodified SPARC binaries with no debugging or relocation information. The coprocessor successfully detected both high-level attacks (directory traversals and cross-site scripting) and low-level memory corruptions (buffer overflows and format string bugs), even in the OS (user/kernel pointer). We can concurrently run all the analyses in Table 3 using 4 tag bits: one for tainting untrusted data, one for identifying legitimate pointers, one for function/system call interposition, and one for protecting the security handler. The security handler is protected by sandboxing its code and data.

We used the pointer injection policy used in [9] for catching low-level attacks. This policy uses two tag bits, one for identifying all the legitimate pointers in the system, and another for identifying tainted data. The invariant enforced is that tainted data cannot be dereferenced, unless it has been deemed to be a legitimate pointer. This analysis is very powerful, and has been shown to reliably catch low-level attacks such as buffer overflows, and user/kernel pointer dereferences, in both userspace and kernelspace, without any false positives [9]. Due to space constraints, we refer the reader to related work for an in-depth discussion of security policies [9].

Note that the security policies used to evaluate our coprocessor are stronger than those used to evaluate other DIFT architectures, including FlexiTaint [5, 7, 24, 26]. For instance, FlexiTaint does not detect code injection attacks and suffers from false positives and negatives on memory corruption attacks. Overall, the coprocessor provides software with exactly the same security features and guarantees as the Raksha design [8, 9], proving that our delayed synchronization model does not compromise on security.

5.2 Performance Evaluation

5.2.1 Performance Comparison

We measured the performance overhead due to the DIFT coprocessor using the SPECint2000 benchmarks. We ran each program twice, once with the coprocessor disabled and once with the coprocessor performing DIFT analysis (checks and propagates using taint bits). Since we do not launch a security attack on these benchmarks, we never transition to the security monitor (no security exceptions). The overhead of any additional analysis performed by the monitor is not affected when we switch from an integrated DIFT approach to the coprocessor-based one.

Figure 3 presents the performance overhead of the coprocessor configured with a 512-byte tag cache and a 6-entry queue (the default configuration), over an unmodified Leon. The integrated DIFT approach of Raksha has the same performance as the base design since there are no additional stalls [8]. The average performance overhead due to the DIFT coprocessor for the SPEC benchmarks is 0.79%. The negligible overheads are almost exclusively due to memory contention between cache misses from the tag cache and memory traffic from the main processor.

Program (Lang)	Attack	Analysis	Detected Vulnerability
gzip (C)	Directory traversal	String tainting + System call interposition	`Open` file with tainted absolute path
tar (C)	Directory traversal	String tainting + System call interposition	`Open` file with tainted absolute path
Scry (PHP)	Cross-site scripting	String tainting + System call interposition	Tainted HTML output includes $< script >$
htdig (C++)	Cross-site scripting	String tainting + System call interposition	Tainted HTML output includes $< script >$
polymorph (C)	Buffer (stack) overflow	Pointer injection	Tainted code pointer dereference (return address)
sendmail (C)	Buffer (BSS) overflow	Pointer injection	Tainted data pointer dereference (application data)
quotactl syscall (C)	User/kernel pointer dereference	Pointer injection	Tainted pointer to kernelspace
SUS (C)	Format string bug	String tainting + Function call interposition	Tainted format string specifier in syslog
WU-FTPD (C)	Format string bug	String tainting + Function call interposition	Tainted format string specifier in `vfprintf`

Table 3: The security experiments performed with the DIFT coprocessor.

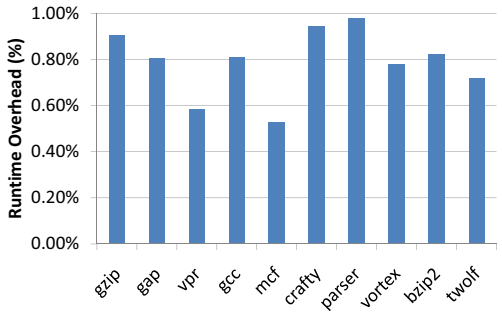

Figure 3: Execution time normalized to an unmodified Leon.

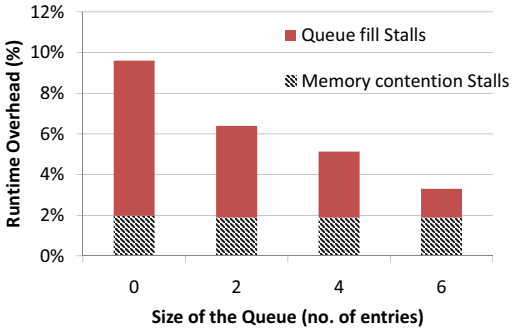

Figure 4: The effect of scaling the size of the decoupling queue on a worst-case tag initialization microbenchmark.

We performed an indirect comparison between the coprocessor and multi-core approaches for DIFT, by evaluating the impact of communicating traces between application and analysis cores, on application performance. To minimize contention, the multi-core architecture [3] uses a 32-Kbyte table for value prediction, that compresses 16 bytes of data per executed instruction, to a 0.8 byte trace. We found the overhead of exchanging these compressed traces between cores in bulk 64-byte transfers to be 5%. The actual multi-core system may have additional overheads due to the synchronization of the application and analysis cores.

Since we synchronize the processor and the coprocessor at system calls, and the coprocessor has good locality with a small tag cache, we did not observe a significant number of memory contention or queue related stalls for the SPECint2000 benchmarks. We evaluated the worst-case scenario for the tag cache, by performing a series of continuous memory operations designed to miss in the tag cache, without any intervening operations. This was aimed at increasing contention for the shared memory bus, causing the main processor to stall. We found that tag cache misses were rare with a cache of 512 bytes or more, and the overhead dropped to 2% even for this worst-case scenario. We also wrote a microbenchmark to stress test the performance of the decoupling queue. This worst-case scenario microbenchmark performed continuous operations that set and retrieved memory tags, to simulate tag initialization. Since the coprocessor instructions that manipulate memory tags are treated as `nops` by the main core, they impact the performance of only the coprocessor, causing the queue to stall. Figure 4 shows the performance overhead of our coprocessor prototype as we run this microbenchmark and vary the size of the decoupling queue from 0 to 6 entries. For these runs we use a 16-byte tag cache in order to increase the number of tag misses and put pressure on the decoupling queue. Without decoupling, the coprocessor introduces a 10% performance overhead. A 6-entry queue is sufficient to drop the performance overhead to 3%. Note that the overhead of a 0-entry queue is equivalent to the overhead of a DIVA-like design which performs DIFT computations within the core, in additional pipeline stages prior to instruction commit.

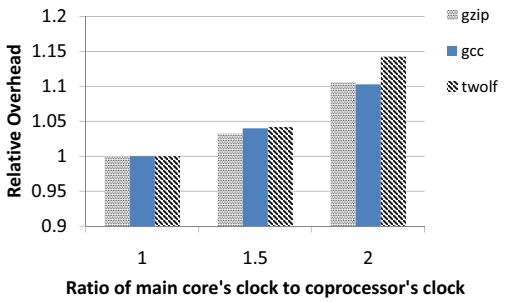

Figure 5: Performance overhead when the coprocessor is paired with higher-IPC main cores. Overheads are relative to the case when the main core and coprocessor have the same clock frequency.

5.2.2 Processor/Coprocessor Performance Ratio

The decoupling queue and the coarse-grained synchronization scheme allow the coprocessor to fall temporarily behind the main core. The coprocessor should however, be able to match the long-term IPC of the main core. While we use a single-issue core and coprocessor in our prototype, it is reasonable to expect that a significantly more capable main core will also require the design of a wider-issue coprocessor. Nevertheless, it is instructive to explore the right ratio of performance capabilities of the two. While the main core may be dual or quad issue, it is unlikely to frequently achieve its peak IPC due to mispredicted instructions, and pipeline dependencies. On the other hand, the coprocessor is mainly limited by the rate at which it receives instructions from the main core. The nature of its simple operations allows it to operate at high clock frequencies without requiring a deeper pipeline that would suffer from data dependency stalls. Moreover, the coprocessor only handles committed instructions. Hence, we may be able to serve a main core with peak IPC higher than 1 with the simple coprocessor pipeline presented.

To explore this further, we constructed an experiment where we clocked the coprocessor at a lower frequency than the main core. Hence, we can evaluate coupling the coprocessor with a main core that has a peak instruction processing rate 1.5x, or 2x that of the coprocessor. As Figure 5 shows, the coprocessor introduces a modest performance overhead of 3.8% at the 1.5x ratio and 11.7% at the 2x ratio, with a 16-entry decoupling queue. These overheads are likely to be even lower on memory or I/O bound applications. This indicates that the same DIFT coprocessor design can be (re)used with a wide variety of main cores, even if their peak IPC characteristics vary significantly.

6 Conclusions

We presented an architecture that provides hardware support for dynamic information flow tracking using an off-core, decoupled coprocessor. The coprocessor encapsulates all state and functionality needed for DIFT operations and synchronizes with the main core only on system calls. This design approach drastically reduces the cost of implementing DIFT: it requires no changes to the design, pipeline and layout of a general-purpose core, it simplifies design and verification, it enables use with in-order cores, and it avoids taking over an entire general-purpose CPU for DIFT checks. Moreover, it provides the same guarantees as traditional hardware DIFT implementations. Using a full-system prototype, we showed that the coprocessor introduces a 7% resource overhead over a simple RISC core. The performance overhead of the coprocessor is less than 1% even with a 512-byte cache for DIFT tags. We also demonstrated in practice that the coprocessor can protect unmodified software binaries from a wide range of security attacks.

7 Acknowledgments

We would like to thank Jiri Gaisler, Richard Pender, and Gaisler Research in general for their invaluable assistance with the prototype development. We would also like to thank Shih-Lien Lu and the anonymous reviewers for their feedback on the paper. This work was supported by an Intel Foundation Graduate Fellowship, a Stanford Graduate Fellowship funded by Sequoia Capital, and NSF awards CCF-0701607 and CCF-0546060.

References

[1] AMD. *AMD Lightweight Profiling Proposal*, 2007.
[2] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *the Proc. of the 32nd MICRO*, Haifa, Israel, Nov. 1999.
[3] S. Chen, B. Falsafi, et al. Logs and Lifeguards: Accelerating Dynamic Program Monitoring. Technical Report IRP-TR-06-05, Intel Research, Pittsburgh, PA, 2006.
[4] S. Chen, M. Kozuch, et al. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *the Proc. of the 35th ISCA*, Beijing, China, June 2008.
[5] S. Chen, J. Xu, et al. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *the Proc. of the 35th DSN*, Yokohama, Japan, June 2005.
[6] J. Chung, M. Dalton, et al. Thread-Safe Dynamic Binary Translation using Transactional Memory. In *the Proc. of the 14th HPCA*, Salt Lake City, UT, Feb. 2008.
[7] J. R. Crandall and F. T. Chong. MINOS: Control Data Attack Prevention Orthogonal to Memory Model. In *the Proc. of the 37th MICRO*, Portland, OR, Dec. 2004.
[8] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *the Proc. of the 34th ISCA*, San Diego, CA, June 2007.
[9] M. Dalton, H. Kannan, and C. Kozyrakis. Real-World Buffer Overflow Protection for Userspace and Kernelspace. In *the Proc. of the 17th Usenix Security Symposium*, San Jose, CA, July 2008.
[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *ACM Communications*, 20(7), 1977.
[11] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *the Proc. of the 11th NDSS*, San Diego, CA, Feb. 2004.
[12] Intel Virtualization Technology (Intel VTx). `http://www.intel.com/technology/virtualization`.
[13] T. Jim, M. Rajagopalan, et al. System call monitoring using authenticated system calls. *IEEE Trans. on Dependable and Secure Computing*, 3(3):216–229, 2006.
[14] LEON3 SPARC Processor. `http://www.gaisler.com`.
[15] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors – a survey. *IEEE Trans. on Computers*, 37(2), 1988.
[16] V. Nagarajan, H.Kim, et al. Dynamic Information Tracking on Multcores. In *the Proc. of the 12th INTERACT*, Salt Lake City, UT, Feb. 2008.
[17] N. Nakka, Z. Kalbarczyk, et al. An Architectural Framework for Providing Reliability and Security Support. In *the Proc. of the 34th DSN*, Florence, Italy, 2004.
[18] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *the Proc. of the 12th NDSS*, San Diego, CA, Feb. 2005.
[19] Perl taint mode. `http://www.perl.com`.
[20] F. Qin, C. Wang, et al. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *the Proc. of the 39th MICRO*, Orlando, FL, Dec. 2006.
[21] M. Rajagopalan, M. Hiltunen, et al. Authenticated System Calls. In *the Proc. of the 35th DSN*, Yokohama, Japan, June 2005.
[22] W. Shi, J. Fryman, et al. InfoShield: A Security Architecture for Protecting Information Usage in Memory. In *the Proc. of the 12th HPCA*, Austin, TX, 2006.
[23] Personal communication with Shih-Lien Lu, Senior Prinicipal Researcher, Intel Microprocessor Technology Labs, Hillsboro, OR.
[24] G. E. Suh, J. W. Lee, et al. Secure Program Execution via Dynamic Information Flow Tracking. In *the Proc. of the 11th ASPLOS*, Boston, MA, Oct. 2004.
[25] N. Vachharajani, M. J. Bridges, et al. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *the Proc. of the 37th MICRO*, Portland, OR, Dec. 2004.
[26] G. Venkataramani, I. Doudalis, et al. FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation. In *the Proc. of the 14th HPCA*, Salt Lake City, UT, Feb. 2008.
[27] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *the Proc. of the 15th USENIX Security Symp.*, Vancouver, Canada, Aug. 2006.