WHISK: An Uncore Architecture for Dynamic Information Flow Tracking in Heterogeneous Embedded SoCs *

Joël Porquet and Simha Sethumadhavan
Department of Computer Science, Columbia University, NY, NY 10027
E-mail: simha@cs.columbia.edu

ABSTRACT

In this paper, we describe for the first time, how Dynamic Information Flow Tracking (DIFT) can be implemented for heterogeneous designs that contain one or more on-chip accelerators attached to a network-on-chip. We observe that implementing DIFT for such systems requires holistic platform level view, *i.e.,* designing individual components in the heterogeneous system to be capable of supporting DIFT is necessary but not sufficient to correctly implement full-system DIFT. Based on this observation we present a new system architecture for implementing DIFT, and also describe wrappers that provide DIFT functionality for third-party IP components. Results show that our implementation minimally impacts performance of programs that do not utilize DIFT, and the price of security is constant for modest amounts of tagging and then sub-linearly increases with the amount of tagging.

Categories and Subject Descriptors

C.0 [**General**]: Hardware-Software Interfaces; D.4.6 [**Operating Systems**]: Security and protection—*Information flow controls*

General Terms

Security, Simulation, Evaluation, Performance

Keywords

Security, System-on-Chip, Network-on-Chip, Heterogeneous designs, Hardware accelerators, Dynamic Information Flow Tracking

*This work was supported by grants FA 99500910389 (AFOSR), FA 865011C7190 (DARPA), FA 87501020253 (DARPA), CCF/TC 1054844 (NSF), Alfred P. Sloan fellowship, and gifts from Microsoft Research, WindRiver Corp, Xilinx and Synopsys Inc. Any opinions, findings, conclusions and recommendations do not reflect the views of the US Government or commercial entities.

International Conference on Hardware/Software Codeisng and System Synthesis, September 29 - October 4 2013, Montreal, Canada

1. INTRODUCTION

Dynamic Information Flow Tracking (DIFT) is a valuable system primitive that finds widespread use in security, privacy, and program analysis applications. For example, DIFT has been used to ensure that private data does not leave a smart phone, detect security attacks such as SQL injection or buffer overflows or identify fault locations in programs when they fail [1, 2]. To support DIFT in a computing system each data item in a program is enhanced to include a tag that identifies some property of that data item. Then during program execution, as old data items are modified the properties of their tags are also modified, or as new data items are produced they get new property tags according to some DIFT policy. The specific policy for creating and propagating tags is based on how DIFT is used: in a privacy application, for instance, data from the GPS receiver may be tagged as confidential, and this data and derivatives may be unsafe to leave the phone through any network interface. The size of the tags used in DIFT can vary widely depending on the application, and range from 1-bit taint tags for security to multi-byte object tags that specify data type or object evanescence [3].

The central question we address in this paper is: *What SoC platform architecture will allow us to easily integrate DIFT support?*

The question of platform architecture for DIFT has not been addressed previously. The main focus so far in the DIFT research area has been how to enhance processor architectures with DIFT [4, 5, 6, 7, 8]. In this paper we ask how we can design the DIFT mechanism at the platform level so that it is simple for third-party IP components, be it accelerators, controllers or even special cores, to be easily integrated in the SoC without intrusive changes. Towards this goal, we provide a set of recommendations for platform designers to implement DIFT as a general hardware service.

We will explain the capability we wish to provide with a simple but realistic example. Let us say we want to build a SoC with DIFT support. Assume that our SoC only has a general-purpose core and a controller, say a DMA engine. Let us also say that on this system the data and tags for the data are stored in different locations in DRAM memory (for efficiency reasons). If the DMA engine is unaware of the separation of tags and data it will miss the tags associated with the data during copies and thus break information flow tracking. Clearly the DMA engine needs to be aware of the tag storage mechanism, *i.e.,* it should know how to compute the address of the tags given the data address. Now, in our simple SoC, instead of a DMA engine, let us say we had a

compression accelerator (or any other computational accelerator that modifies the input data). In addition to being aware of the tag storage, it should also be capable of propagating the tags through the datapath within the accelerator. In this paper, we show how to extend the platform architecture so that any SoC component can easily find the tags stored in memory; the issue of tag propagation is orthogonal and not described in this paper. However, as it will become clear later in the paper for many common third-party IP components, the implementation of tag propagation logic is straightforward, or in some cases may not even require modifications to the accelerator.

To easily integrate third-party IP components in a DIFT aware platform we propose a new architecture called WHISK. In our architecture the data and tags are stored separately in memory to keep a low area overhead and improve flexibility. The salient features of our architecture are (Figure 1):

(a) Implicit Addressing of Tags and Data: We propose an architecture in which a NoC client on the SoC does not have to know anything about the tag layout or storage. Instead of sending a pair of addresses to access a data and its associated tag, which forces the clients to know the association mechanism between data and tags, in WHISK we allow clients to send only the data address and automatically receive or send the requested data along with its associate tag in the same packet. This strategy lowers the complexity of adapting DIFT to IP components since tags are automatically and transparently accessed with data. Further since the tag calculation is isolated from the clients, the system supports flexible tag layout and storage in memory, allowing DIFT to be easily customized for different applications.

(b) Atomic transmission: While the data and tags are stored separately in memory to keep a low area overhead, they are transported together from memory through the interconnect instead of being fetched separately as is done in single processor DIFT implementations. This coupled atomic transport decreases the complexity of adapting accelerators to DIFT by avoiding subtle memory coherence and consistency problems between tags and data.

(c) Pipelined transfer: In our WHISK NoC protocol we send the data from/to memory one cycle after the tag. This has three main benefits. First it reduces the area overhead and design complexity since the data and tags can be sent on the same interconnect. Second the tags are already available at the clients when the data arrives at the client mitigating or completely avoiding serialization latencies during DIFT processing. Finally, since the tag and data use the same interconnect, the tag can be arbitrarily large: it can be as large as the data or if needed even larger by sending the tag over multiple packets. This allows flexible implementations of DIFT policies.

(d) Configurable, multi-granular caching: In DIFT applications, often large portions of nearby data items tend to have the same tag properties. This property can be used to reduce the area overhead of tags by representing common properties for many addresses using one tag instead of one tag per address. WHISK supports this multi-granular tag optimization. Further, in WHISK we allow clients to cache these tags to allow temporal reuse of tags to avoid latency overhead of tag accesses. These caches are also implicitly addressed with data.

(e) Standard wrapper for SoC clients: Finally, and perhaps most importantly, we show how all of the above features – implicit addressing, atomic transmission, pipelined transfer and tag caching – can be built in a way that allows these functions to be wrapped around existing clients in the SoC with minimum changes to the NoC or the SoC memory architecture (Figure 2). Our wrappers also handle OS interrupt processing. The wrappers are placed on the path between the NoC and the clients.

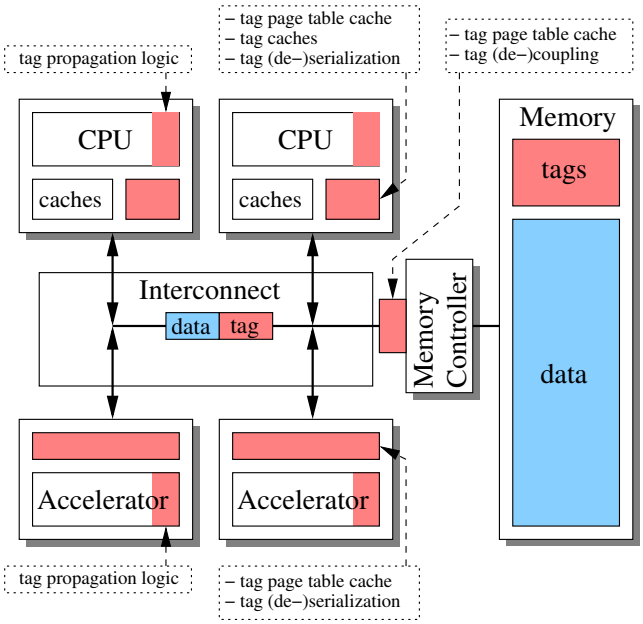


Figure 1: Overview of WHISK: red regions denote tag extensions. The callout boxes describe functionality.

To examine the practicality of WHISK we developed a cycle accurate SoC in SystemC. We were able to integrate different types of accelerators with DIFT into the system (*e.g.,* compression, cryptography). We were also able to boot an embedded Operating System and run full applications. To test the utility of DIFT as a service we measure the impact of DIFT for different amounts of tagging by varying the fraction of the program's input data that can be tagged, and the width of the tags. This is different from prior works where overheads of DIFT were measured for specific applications of DIFT such as buffer overflows. Our experimental results with microbenchmarks show that WHISK exhibits security-proportionality: the performance overhead is relatively proportional to the amount of tagging in the system. When running full software applications, however, the performance overhead stays almost constant, *i.e.,* is less impacted by the amount of tagging, because the cost of WHISK is amortized with microarchitectural optimizations, and also because of tag aggregation and caching. Finally, when active but not used, *i.e.,* when the amount of tagging is null, the overhead of WHISK is negligible.

2. BACKGROUND

There are generally two ways to store tags in DIFT architectures.

Coupled scheme.

In the *coupled scheme* each data element is physically stored with its associated tag [5, 6] and they are always

transmitted atomically throughout the system. This means that the main memory, as well as all the components along the memory chain, the system bus, processors datapaths, caches and registers are all made wider to accommodate the tag. Although this scheme seems to be the easiest solution in terms of implementation complexity, it necessitates special adaptations, as highlighted by Venkatramani *et al.*[7]. For example, this scheme requires use of non-standard memory bank sizes and modification to processor instructions to allow them to access and manipulate the tag bits. Further, tag storage is wasted when tagging is not needed, and new processor instructions are required to access and manipulate tag bits independently (e.g. for initialization). Consequently the area overhead can be tremendous especially given the need for using large tags (from at least 8 bits [9] and up to 32 bits [10, 11]) as the coupled scheme adds a +3% overhead for each additional tag bit (for a word-granularity policy).

Decoupled scheme.

The *decoupled scheme* consists in dedicating a portion of memory to store tags separately from the data. Tags and data are thus both in memory but in two separate regions. The separation involves the use of an association algorithm to find the associated tag of a data when reading or writing this data. Some of the existing approaches [7, 12] store tags as a bitmap in a protected area of applications' virtual address space and perform the association between data and tags via a simple index calculation. Another type of decoupled scheme suggests that tags protect the whole address space by the means of a multi-level page table [4, 13]. For instance, code segments tend to share the same tag for different DIFT applications and as such may not need such a fine tagging granularity. The first level of this table typically covers the address space at page-level granularity: pages can therefore be considered as fully tagged or untagged, or as partially tagged. In the latter case, a second level in the tag page table allows to protect a data page at a finer granularity (e.g. word- or byte-level). Such multi-granular approaches are able to accommodate flexible tagging requirements at runtime, reducing the area overhead but still allowing fine granularity when needed.

Discussion.

In prior implementations the complexity of the decoupled scheme comes mostly from the fact that data and tags must be retrieved separately since they are stored in different memory regions. The association algorithm is usually implemented in the processor core, and generally involves using additional TLBs for tags and tag caches for performance improvements. A software/hardware mechanism is also required to detect and process when tags have to be aggregated: for example, when storing a tagged data element in an untagged page for the first time, the tagging granularity must be refined by creating a second level in the tag page table for that particular data page.

Using a decoupled scheme also raises the possibility of incoherence and inconsistency between tag and data in multicores. In proposals where tags are manipulated in the same pipeline stage as data [4, 6] this issue is avoided by ensuring that when writing or reading a data and its associated tag, both are available in their respective cache and the operation completes in one cycle. For proposals that perform tag manipulation in different pipeline stages than the stage reading/operating on data [7, 12], this issue is more challenging but was addressed by Venkatramani *et al.* [7]. Note that architectures implementing the coupled scheme are not affected by coherence and consistency issues since data and tags are always paired together in the system.

In general DIFT also requires system level support to preserve tags when data leave memory. Crandall *et al.*[5] store the tags in kernel memory when a data page is swapped out on disk so that they can be restored when the data page is swapped back in. Some software implementations [14, 15] modify the file system so that when data is written to a file, tags are stored as well.

DIFT in SoCs.

SoC design issues and requirements have never been considered in past DIFT proposals. Yet, such systems are spreading, and there is strict emphasis on low cost and low design complexity. In our approach, we accommodate these requirements by adopting a hybrid scheme of the coupled and decoupled schemes, where data and tags are transported together between memory and cores and then split just before storage in memory. This hybrid scheme guarantees reading and writing atomicity and thus consistency between data and tags. And tagging granularity refinements are detected at the earliest, by putting small hardware modules in front of the clients. Further many of these changes can be wrapped over existing clients enabling easy integration of security functionality.

3. THE WHISK ARCHITECTURE

In this section we describe our DIFT architecture called WHISK.

Tag storage.

WHISK uses a two-level table for tag storage in DRAM. Both levels are indexed by physical memory address and return the tag associated with that address. The first level is a linear array containing tags for each physical memory page, indicating if the page is fully untagged, fully (uniformly) tagged, or partially tagged. The second level holds tags for pages at word granularity, and can be allocated on demand when a page becomes partially (non-uniformly) tagged. The page table is statically allocated and guaranteed to be present in physical memory, which is realistic for many embedded system-on-chip designs.

Hardware support for tag management.

In WHISK the mechanism for associating data with tags is integrated in a small hardware module in front each network client such as an accelerator or a memory controller (See Figure 2). Conceptually, on a read operation from a client, this module retrieves the requested data at the specified address, and the tag page table is accessed to find the tag corresponding to the given address, before giving it back the client that initiated the request. The same concept is reversed for write operations: if the tags have already been allocated they are updated by the hardware module otherwise the modules issue an interrupt to the OS to allocate tags.

Some clients may optionally include caching for page level tags in the wrappers. These Page Table Caches (PTCs) return for a given memory access whether the containing page

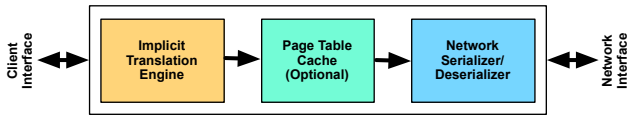


Figure 2: DIFT Wrapper

is fully untagged or tagged (along with the tag value), or partially tagged. In the former two cases, the tags need not be explicitly accessed since the tag property of the requested data is already known. It is only for partially tagged pages that tags might need to be fetched from memory. Such caching can significantly reduce the traffic overhead. Since the tags are present in the wrapper, in WHISK, the DIFT module *implicitly* retrieves the tags when the data is accessed *i.e.,* the clients are not able to identify the address mapping between data and tags.

In WHISK the communication interface is widened with an extra two-bit field which encodes three possible commands. The `NONE` command instructs the memory to return or to expect only the data. The `WITH` command instructs the memory to return or expect a cache line along with its associated tag. The `ONLY` command instructs the memory to return the tags associated to a certain cache line. Since in most cases data is expected to be in fully tagged or untagged pages, a large majority of read or write operations should cause only one memory access by using a `NONE` command and packets will not need to be extended with tags. This strategy reduces the traffic overhead on the interconnect and the number of memory accesses, lowering both the energy consumption and the performance overhead.

Software support for tag management.

There are three mandatory tasks that the OS has to perform. The first task is to allocate the tag page table, usually during the system boot. The second task is to configure the different PTCs of the system, namely the register that holds the base address of the tag page table. The last task concerns writes to untagged pages: in this case, the OS must provide a exception/interruption handler to process those requests, by creating on-demand a second level in the tag page table.

IP Integration in WHISK.

From an implementation standpoint, WHISK introduces wrappers in front of NoC clients (IP blocks), which provide two functions. First, the wrapper abstracts away the decoupled nature of the tags by acting as a serializer/deserializer for write/read operations. Second, by hosting a PTC, the wrapper is able to obviate page table accesses when tags are known (using the `NONE` command), and also to recognize, at the earliest point, when a page refinement is required.

Tag propagation and tagging policies.

Tag management provides the infrastructure for managing tags; another important part of DIFT deals with the tag propagation, that is how cores and accelerators compute and propagate tags through their internal storage location, and the policies for such propagation. While WHISK does not focus on that second part, WHISK is compatible with existing mechanisms [16]. Our design is rather agnostic to those propagation policies as long as data and tags are conceptually read or written atomically in their PTCs by the clients which the wrappers strive to provide.

Parameter	Specification
L0/L1 instruction cache	16KB/64KB, 4-way set associative
L0/L1 data cache	64KB/512KB, 4-way set associative
Cache block size	64 bytes
L0/L1 latency	10 cycles
Interconnect width/latency	32 bits/7 cycles
Coherence protocol	MESI-like directory-based
Memory latency	150 cycles

Figure 3: Architectural and design parameters for the WHISK prototype

4. WHISK IMPLEMENTATION

4.1 Hardware system

We modified SoCLib [17], an open-source library of hardware simulation models described in SystemC to model WHISK. We chose this prototyping framework for its ability to rapidly modify the simulation models to perform design space exploration compared to a classic RTL description, its fast simulation speed and its cycle-accurate bit-accurate characteristic which provides a similar feedback as a RTL implementation in terms of performance measurement.

Our baseline hardware system is based on three hardware accelerators and a single MIPS processor (with CPI of 1). This processor does not perform virtual memory management and is equipped with a two-level write-back cache system. Table 3 summarizes the main architectural and design parameters of our prototype. The processor is modified to implement a simple logical-OR based tagging propagation policy and its interface, from core to cache, is extended with a *tag* field. The tag caches are 1/8 the size of their matching instruction/data caches. For instance, the tag-data L1 cache is a 4-way set-associative cache which counts 32 sets, while the size of the cache block depends on the tag width: 16 bits if the width of tags is 1 bit (1-bit for every word of the 64-byte data block) up to 64 bytes if the width of tags is 32 bits (1 word tag for every data word). The PTCs are 64 entries, 2-way set associative. The data and tags can be stored in different memory banks, therefore allowing data and tag requests to be concurrently scheduled by the memory controller in back to back cycles.

We implement three accelerators: DMA, AES cryptographic accelerator and LZSS compression accelerator. When configured, these accelerators perform transfers to and from memory by repeating the two following operations as many times as required: a burst read (the size of a cache line, to avoid taking over the memory controller for too long) from memory into an internal buffer, and a burst write of the output back to memory. For the DMA and AES accelerators we assume that the output has the same tag as input. For the LZSS accelerator, when tagged symbols cannot be compressed, they are let unmodified in the input along with their tag. When a sequence of symbols can be compressed (*i.e.,* matched with a identical sequence of symbols already encountered and still in the dictionary), the resulting output (*i.e.,* 2 symbols to indicate the offset of the matching pattern and the length of this pattern) are tagged if at least

one of the input symbols is tagged.

4.2 Operating system

We use MutekH [18], a open-source exokernel-based operating system. In this operating system, software applications are compiled with the kernel and run in kernel land, using a POSIX thread library. MutekH is extensible and allows an easy development of new kernel modules to perform design space exploration.

We add a tag module to MutekH to manage our decoupled tagging scheme. This module, which has an approximative size of 700 SLOC, provides an API mainly for:

- Building the initial tag page table. Such table covers the physical memory address space, which is set as completely untagged, at page-level granularity.
- Configuring and enabling the different PTCs of the system, by providing them with the root address of the tag page table.
- Maintaining the tag page table, that is tagging regions of memory, either at page-level granularity or at word-level granularity after performing page refinements.

The code of this new module is mostly self-contained and there are only three locations in MutekH's core where we insert function calls to that module. At system boot, we add a call to the tag page table creation to enable the WHISK architecture. When creating a new application, the memory area used to save the context of the processor in case of interrupts/exception is set as a partially tagged region, to avoid facing an infinite loop of exceptions when possibly saving a tagged register into it for the first time. And finally, we add an exception/interrupt handler to manage page refinement requests at runtime, either coming from the processor or accelerators' PTCs. All of this code can actually be implemented in a dedicated hardware unit to minimize OS dependency even further.

5. EVALUATION

Our goals are two-fold. First we aim to show that, when active but not in use, our security infrastructure has unnoticeable impact on the system's performance. Second, when in use, we seek to characterize its impact according to the amount of tagging. We define this property as "security-proportionality" and interpret the amount of tagging as the combination between the percentage of tagged input data that is supplied to a benchmark and the width of the tags (1, 8, 16 or 32 bits). It is theoretically expected that impact to be higher when the amount of tagging increases since more of the security infrastructure will be used. This approach for measuring is different from prior approaches where one or few uses of DIFT is used to characterize the overheads.

For this evaluation, we use three sets of benchmarks. Since our work primarily focuses on heterogeneous systems, the first set includes microbenchmarks to evaluate the security proportionality of WHISK between hardware accelerators and memory. These benchmarks make use of three accelerators chosen to represent a range of existing accelerators in terms of internal organization and access patterns. The first accelerator is a simple DMA transfer engine; the second accelerator implements an AES engine, which is basically based on the DMA engine but adds a non-negligible processing latency; finally, the third accelerator is a LZSS compression engine, which adds the characteristic of being stateful across transactions. The second set of benchmarks is composed of software applications. We choose multimedia-oriented and very data intensive applications to emulate a typical workload in some embedded systems. The multimedia software applications are: JPEG encoder (cjpeg), MP3 player (minimad), Xvid decoder and encoder (xvid_dec and xvid_enc). We make the working sets of the two latter applications vary by using different image sizes (QCIF and CIF). Finally we include a software application that combines accelerators and general purpose cores to study the performance/security tradeoffs.

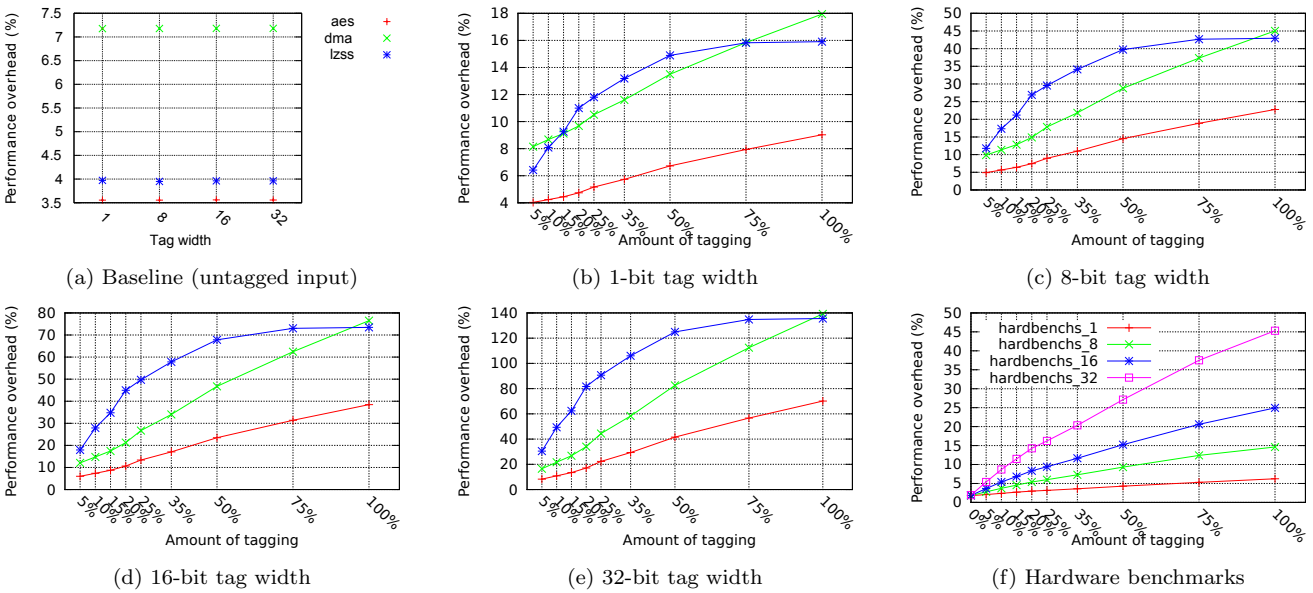
The tagging status of the inputs is configurable. Given that the input data span across many pages, it is possible to define the ratio of those pages that are set as partially tagged (*i.e.,* tagged at the granularity of words), and within those pages, the ratio of words that are tagged (*i.e.,* with a tag different than 0).

5.1 Microbenchmark Results

Figure 4a shows the performance overhead induced by WHISK when the input data to each of the benchmarks is completely untagged at page-level granularity. The overhead for DMA, AES and LZSS is respectively 7.25%, 3.5% and 4% and is constant whatever the width of the tagging. DMA has the biggest overhead since this accelerator does not perform any internal processing which, in the case of AES and LZSS, helps hiding part of the overhead of WHISK.

The origin of this 7.25% overhead is mainly due to write operations from the accelerator. Read operations are fairly transparent: data packets coming from memory are directly transmitted to the accelerator without any buffering in the wrapper; there are a couple of cycles to start this transmission within the tag wrapper but they are negligible compared to the memory latency (about 320 cycles for read operations at the accelerator). For write operations however, buffering data bursts coming from the accelerator is mandatory. If the targeted memory page is fully untagged (which is always the case for this baseline evaluation) and one of the word produced by the accelerator is tagged, then the wrapper must interrupt the OS in order to request a page refinement and switch the targeted page to word-granularity. It would be too late to detect this need for refinement on the fly while transmitting the write burst to memory. In our implementation, buffering write operations takes about 20 cycles: 16 cycles to buffer the 16 words that compose a write burst, and a few extra cycles to drive the few FSMs inside the wrapper. Those 20 cycles are a non-negligible impact on performance because the latency to memory is smaller for write operations (about 60 cycles from the point of view of accelerators) than for read operations. When the memory controller receives a write operation, the operation is acknowledged immediately to the initiator core, making the actual write to memory only a "hidden" post-latency to the operation.

The Figures 4b,4c,4d,4e show the results of the second set of experiments, when the ratios of tagging are progressively raised to a fully tagged input. In those experiments, only the ratio of partially tagged pages is increased, from 5% of partially tagged up to 100%, and all the words within those pages are tagged (the percentage of tagged words does not impact the results). Unsurprisingly the cost of security



(a) Baseline (untagged input)

(b) 1-bit tag width

(c) 8-bit tag width

(d) 16-bit tag width

(e) 32-bit tag width

(f) Hardware benchmarks

Figure 4: Security proportionality for hardware accelerators

increases along with the amount of tagging. The resulting overheads are higher than for software applications because there is no caching within the tag wrappers, which means the tags have to be transmitted along each request targeting a partially tagged page (both for read or write operations). This property also explains why the overhead is clearly higher when using 32-bit wide tag, since in such case, the memory accesses are doubled in size (16 bytes of data concatenated with 16 bytes of tags). The cost of refinement requests to the main OS is also not negligible especially when the percentage of partially tagged input pages is high (about 3000 cycles for processing a page refinement request by the OS).

For this set of experiments, we added an optimization to the tag wrappers which lowered the overhead by several percents. The idea was to pipeline write operations as often as possible, that is to start sending the data written by the accelerator to memory without buffering them in the tag wrapper. As previously explained, although non-buffering is impossible when the output memory location is tagged at page-level granularity, it is possible when the output page is marked as partially tagged, because then, the wrapper can leverage the fact no refinement request will be needed (the refinement has already been performed before). Typically, when 100% of the input is tagged, then 1.5% of the output write operations needs to be buffered because a page refinement is expected, but the 98.5% of the remaining write operations can be pipelined without buffering.

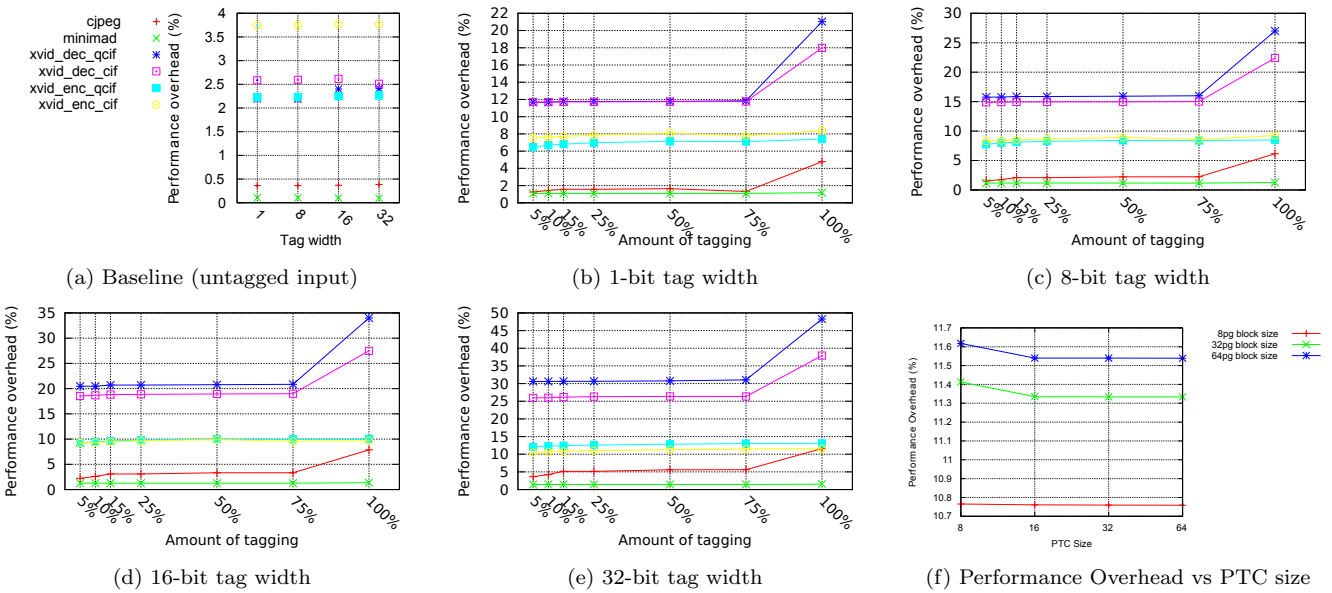
We also run another experiment using all the microbenchmarks simultaneously. In this third experiment, the three accelerators are instructed with transfers at the same time, in order to test how the WHISK architecture is able to support a higher load. Since AES is the benchmark that takes the longest time, we loop the two others five times so all of them finish approximately at the same time, and the infrastructure endures a constant load across the length of the simulation. Figure 4f shows the resulting performance overheads for different tagging ratios. We observe that those overheads are smaller than for the previous experiments when each benchmarks are tested separately. This phenomenon can be explained by the fact that we do not reinitialize the output pages between the five runs of DMA and LZSS, so after the first run, the output pages are already set as partially tagged pages which avoids further costly refinement requests and thus lowers the overhead.

5.2 Software Application Results

Figure 5a shows the baseline performance overhead of WHISK. We observe the overhead to be quite negligible, namely in the range of 0% and 3.75%, and is stable regardless of the width of tagging. This overhead comes mainly from the accesses to the tag page table by the PTCs of the system (in the processor and in the memory controller). Processing smaller data working-sets, cjpeg and minimad have the lowest overhead which means fewer accesses from the PTCs, while the four versions of xvid_dec and xvid_enc have a bigger overhead since they all process bigger data working-sets.

In the case when WHISK is completely inactive, that is the tag page table is not built at boot-time nor the PTCs of the system are initialized, then there is no performance overhead for software applications (the microbenchmarks still slightly suffer from the tag wrappers latency as mentioned earlier).

In the second set of experiments, we vary the ratios of tagged input, from 5% to 100% of input pages set as tagged. In Figures 5b,5c,5d,5e, we see that the performance overhead is fairly constant with respect to the examined benchmark application and the width of tags. It is only when 100% of the input pages are set as tagged that the overhead stops being constant for most of the benchmark applications and increases. We explain this behavior first by the number of refinement requests that needs to be performed (when writing tagged output words for the first time in fully untagged output pages), and most of all by misses in the tag caches which are 1/8 smaller than their data counterparts.



(a) Baseline (untagged input)

(b) 1-bit tag width

(c) 8-bit tag width

(d) 16-bit tag width

(e) 32-bit tag width

(f) Performance Overhead vs PTC size

Figure 5: Security proportionality for software applications

In a third experiment, we vary the size of the PTCs to observe their impact on an application which accesses common tagged data on both the CPU and accelerators. We simulate an application which performs a hardware accelerated video encoding while simultaneously performing a JPEG image compression on the CPU, then combine both output buffers into a compressed buffer using a hardware accelerated compressor, and finally compute the SHA-512 hash of the compressed buffer using the CPU. Figure 5f shows the performance impact at various PTC sizes for input buffer sizes of 8, 32, and 64 pages.

6. RELATED WORK

While there has been a lot of work on DIFT at the processor architecture level there is no work at the platform level. Our work addresses this gap. Our work makes it easier to build secure embedded systems which are seriously threatened today.

In 2004, Suh *et al.*[4] proposed one of the first hardware DIFT architectures for addressing software attacks. In their architecture, the tag is a taint bit that can be applied to memory elements as small as bytes. Within the processor, the propagation policy distinguishes four categories of instructions providing simple rules for each one (*e.g.,* for a computation-copy instruction, the taint of the result receives the taint of the operand). Whenever a tainted data is used as an execution address, which is likely to characterize a memory corruption attack, the processor raises a security exception. In order to circumvent the overhead of tagging every byte of memory, they introduced a multi-granularity policy: a page table structure allows to tag data blocks as large as entire pages or quadwords or words with only one taint bit. It is only upon the first write operation on a data smaller than the granularity of the page it belongs to, that the OS refines the tagging granularity for this page. Their mechanism relies on TLBs to be extended in order to cache the tag type of memory pages. Finally, to avoid extending each processor cache block with tag bits, they introduced separate tag caches.

The same year, Crandall *et al.*[5] proposed Minos also addressing software attacks. Their approach is quite similar to Suh *et al.*[4] with a noticeable exception that they did not try to reduce the memory overhead induced by the tag bit, arguing that Moore's law could easily absorb the overhead overtime. In this design, the memory, the common data bus and the whole memory chain of the processor (*i.e.,* from caches to registers) are thus extended with tag bits at the granularity of word.

Mostly based on both these approaches, the DIFT concept has subsequently been adapted following different axis, but the main focus has been toward less intrusive hardware implementations and more flexible propagation policies.

Raksha [6] was the first approach targeting flexibility. The implementation mostly resembles the previous hardware DIFT architectures, but it supports up to four programmable, independent and concurrent security policies each with its own set of rules for propagation and checks. The software is able to control each policy by using pre-existing rules or defining custom rules for each class of instructions. The flexibility provided by this scheme allows to detect various type of software attacks, from high-level attacks (*e.g.,* SQL injection) to low-level attacks (*e.g.,* memory corruption and format strings).

Flexitaint [7] is another approach mostly targeting flexibility, but also aiming to reduce the hardware impact on out-of-order processor architectures. Instead of placing the taint tag propagation logic along with the regular computation, they introduce an additional stage at the back-end of the pipeline leaving the other stages mostly unmodified. In this new stage, the propagation computation is either performed by software customizable rules or fixed-rules. Taints are stored as a packed array in a protected area of the virtual address space of each program. This organization avoids the need to modify the memory chain as well as the bus and allows a full compatibility with conventional OS mechanisms (*e.g.,* disk swapping, copy-on-write, etc.), but an additional

Name	Year	Tag width/Granularity	Tag management	Multiprocessor	Accelerator support	Target
Suh/DIFT [4]	2004	1/byte	decoupled (page-table)	Implicit MP support	No	Non- and control data attacks
Minos [5]	2004	1/word	coupled	Implicit MP support	No	Control data attacks
RIFLE [19]	2004	unspecified	unspecified	No	No	Information leakage
Raksha [6]	2007	4/word	coupled	Implicit MP support	No	High- and low-level attacks
Flexitaint [7]	2008	1-2/word	decoupled (virtual memory)	MP support	No	Unspecified
SHIFT [12]	2008	1/byte-quadword	decoupled (virtual memory)	No	No	High- and low-level attacks
Kannan/Coprocessor [13]	2009	4/word	decoupled (page-table)	No	No	Same as Raksha
Deng/FPGA [20]	2010	1-8/word	decoupled (page-table)	No	No	Flexible security policies
SIFT [21]	2011	unspecified (1)	unspecified	No MP support	No	Unspecified (attacks)
WHISK	2013	1-32/word	decoupled (page-table)	Yes	Yes	Heterogeneous systems

Figure 6: Summary of previous works

L1 cache is introduced within the processor to cache taints.

Two other approaches leverage existing processor architecture features to alleviate the implementation of DIFT. SIFT [21] dedicates one spare thread of a SMT processor to perform taint propagation and checks, and generates DIFT instructions at the commit stage of the pipeline to feed this new security thread. SHIFT [12] uses the speculative execution and deferred exceptions mechanisms which exist in some processor architectures (*e.g.,* Itanium).

Another effort to avoid modifications to the design of processors has been made by Kannan *et al.*[13]. They propose an off core coprocessor for DIFT (based on Raksha [6]) thus completely decoupling the DIFT functionality (tainting state - registers and caches -, propagation and checks) from the regular computation of the processor. To improve flexibility, Deng *et al.*[20] suggested that such a coprocessor should be implemented on an on-chip reconfigurable fabric, tightly coupled with the main processing core. With respect to those off-core architecture, Kannan [22] proposed a mechanism for such a decoupled approach to keep the proper order of memory accesses.

Unlike the previous approaches, RIFLE [19] focuses on the detection of information leakage. This architecture is composed of two steps. First, an original program is transformed using a binary translation mechanism. This translation has two goals: converting all implicit flows to explicit flows, and changing the ISA used by the program from the regular ISA to an information-flow security ISA (*i.e.,* adding taint management). Second, the translated program runs on a processor architecture that supports a conventional implementation of DIFT.

Table 6 presents a summary of those architecture-level approaches.

Another direction has been explored, to implement DIFT in hardware, at a level lower than the architectural level. Tiwari et al. [8] propose GLIFT, an approach for tracking information-flow at gate-level. GLIFT shows how to generate shadow flow tracking logic for each simple gate in a design, and eventually suggests composition rules between gates to handle more complex structures.In the same scope, Li et al. [23] describe a new hardware description language, named Caisson, which allows to construct information-flow secure designs. Then using existing synthesis tools, the resulting secure design is competitive with a equivalent insecure design and significantly better than a GLIFT-enhanced design. Both approaches covers tag propagation not management.

Table 6 presents a summary of previously described approaches to DIFT. No previously proposed designs cover seamless integration for SoC based designs.

7. CONCLUSION

Past research has studied the implementation of DIFT in a few contexts (*i.e.,* uniprocessor and homogeneous multiprocessor), under several forms (*e.g.,* integrated to the computational path of processors or part of external dedicated cores) and addressing different aspects (*e.g.,* performance overhead improvements, less intrusive approaches, better flexibility). However, the integration of third party IP blocks like accelerators and controllers in a DIFT architecture had never been considered before, although the use of heterogeneous SoCs is widespread and growing.

In this paper, we presented WHISK, a DIFT architecture which relies on a new tag management layout to better accommodate the integration of hardware accelerators. WHISK adopts a decoupled tag storage, where tags are split from data in memory, but data and tags always travel coupled on-chip. Those architectural choices allow to keep both a low area overhead concerning the tag storage and a low complexity to adapt cores for DIFT by guaranteeing the atomicity of data with their associated tag and easing the access to tags by cores. These changes allow DIFT to be used as a transparent system security primitive for heterogeneous systems.

8. REFERENCES

[1] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[2] M. Dalton, *The Design and Implementation of Hardware Systems for Information Flow Tracking*. PhD thesis, Stanford University, 2009.

[3] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos:limiting mobile data exposure with idle eviction," in *Proceedings of the 10th USENIX conference on Operating systems design and implementation*, OSDI'12, (Berkeley, CA, USA), pp. 77–91, USENIX Association, 2012.

[4] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, vol. 39 of *ASPLOS-XI*, (New York, NY, USA), pp. 85–96, ACM, ACM, 2004.

[5] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, (Washington, DC, USA), pp. 221–232, IEEE Computer Society, IEEE Computer Society, 2004.

[6] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, (New York, NY, USA), pp. 482–493, ACM, ACM, 2007.

[7] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 173–184, IEEE, 2008.

[8] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, (New York, NY, USA), pp. 109–120, ACM, 2009.

[9] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, (New York, NY, USA), pp. 121–132, ACM, 2012.

[10] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood, "A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, (Washington, DC, USA), pp. 94–105, IEEE Computer Society, 2008.

[11] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, (Berkeley, CA, USA), pp. 225–240, USENIX Association, 2008.

[12] H. Chen, X. Wu, L. Yuan, B. Zang, P. Yew, and F. Chong, "From speculation to security: Practical and efficient information flow tracking using speculative hardware," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 401–412, 2008.

[13] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pp. 105 –114, 29 2009-july 2 2009.

[14] A. Ermolinskiy, S. Katti, S. Shenker, L. L. Fowler, and M. McCauley, "Towards practical taint tracking," Tech. Rep. UCB/EECS-2010-92, EECS Department, University of California, Berkeley, Jun 2010.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[16] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in i2c and usb," in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 254 –259, june 2011.

[17] The SoCLib project. `http://www.soclib.fr`.

[18] The MutekH project. `http://www.mutekh.org`.

[19] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, (Washington, DC, USA), pp. 243–254, IEEE Computer Society, 2004.

[20] 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 *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 137–148, IEEE Computer Society, 2010.

[21] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri, "SIFT: A Low-Overhead Dynamic Information Flow Tracking Architecture for SMT Processors," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, (New York, NY, USA), pp. 37:1–37:11, ACM, 2011.

[22] H. Kannan, "Ordering decoupled metadata accesses in multiprocessors," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 381–390, ACM, 2009.

[23] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, (New York, NY, USA), pp. 109–120, ACM, 2011.