Appears in the Proceedings of the
14th IEEE International Symposium on High-Performance Computer Architecture (HPCA), February 2008.

FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation

Guru Venkataramani, *Georgia Tech*, *guru@cc.gatech.edu*
Ioannis Doudalis, *Georgia Tech*, *idoud@cc.gatech.edu*
Yan Solihin, *NC State University*, *solihin@ece.ncsu.edu*
Milos Prvulovic, *Georgia Tech*, *milos@cc.gatech.edu*

Abstract

This paper presents FlexiTaint, a hardware accelerator for dynamic taint propagation. FlexiTaint is implemented as an in-order addition to the back-end of the processor pipeline, and the taints for memory locations are stored as a packed array in regular memory. The taint propagation scheme is specified via a software handler that, given the operation and the sources' taints, computes the new taint for the result. To keep performance overheads low, FlexiTaint caches recent taint propagation lookups and uses a filter to avoid lookups for simple common-case behavior. We also describe how to implement consistent taint propagation in a multi-core environment. Our experiments show that FlexiTaint incurs average performance overheads of only 1% for SPEC2000 benchmarks and 3.7% for Splash-2 benchmarks, even when simultaneously following two different taint propagation policies.

1. Introduction

Software debugging and verification are becoming increasingly complex, and many bugs are also vulnerabilities that can be used as security exploits. To help deal with these problems, a variety of runtime checking and tracking approaches have been proposed. A number of these proposals have adopted dynamic taint propagation, or *tainting*. Typically, a tainting scheme associates a *taint* with every data value. The taint is usually a one-bit field that tags the value as safe (untainted) or unsafe (tainted). Data from trusted sources starts out as untainted, whereas data from an untrusted source (e.g. network) starts out as tainted. Taints are then propagated as values are copied or used in computation. To detect potential attacks, a tainting scheme looks for unsafe uses of tainted values. For example, using a tainted value as a jump target address is considered unsafe because such a jump may allow the attacker to hijack the control flow of the application.

Software taint propagation schemes were proposed as a comprehensive solution against specific types of attacks [5, 10, 11, 20, 23]. However, software-only schemes have large performance overheads and have significant problems with self-modifying code, JIT compilation, and multithreaded applications. Hardware-assisted schemes try to avoid these drawbacks [2, 7, 8, 18].

Many hardware tainting schemes suffer from two problems that limit their practicality. First, their implementation requires non-standard commodity components and a redesign of the entire processor core. Since taint bits are added to every value in memory and in the processor, wider memory, registers, buses and bypasses are needed. The second problem is limited flexibility in specifying taint propagation rules, and in the number of taint bits associated with a value. Most schemes provide single-bit taints with little or no flexibility in how that taint is propagated. Those that allow multi-bit taints do so by significantly increasing the implementation cost, as memory modules, buses, and the processor core must be designed to accommodate the largest supported taint. Similarly, schemes that provide some flexibility in specifying taint propagation rules are still limited to simple rules that mainly target only one particular use of tainting - tainting of inputs to detect attacks.

This paper introduces FlexiTaint, a programmable accelerator for taint propagation. Instead of directly implementing a specific set of tainting policies, FlexiTaint is programmed at runtime to efficiently follow a desired tainting policy. This allows a FlexiTaint-equipped system to implement radically different taint propagation policies for different applications, to upgrade its taint propagation policies as new attacks are devised to circumvent existing policies, and even to use tainting for uses other than attack detection/avoidance. As an accelerator, FlexiTaint is not a comprehensive security solution by itself, but rather a proof-of-concept hardware substrate that can speed up an important class of security solutions, namely taint propagation.

One of the main advantages of taking the programmability approach for taint propagation reduced risk of obsolescence. Whereas a software tool can quickly be upgraded to guard against new attacks, hardware based tools can only be upgraded by replacing the processor or the entire system. This problem is exacerbated by the need to maintain backward compatibility with existing software – once a hardware mechanism is implemented in a processor, new processors must continue to support that functionality. As a result, a hardware scheme can continue to increase the cost and complexity of systems for years, long after attackers have discovered how to circumvent it. This concern makes it very risky to directly implement any specific taint propagation policy or set of policies in a commodity processor. Our programmable accelerator approach allows taint propagation policies to be changed by software as new attacks are devised, reducing the risk of hardware becoming obsolete.

Another advantage of FlexiTaint is that it decouples taint storage and processing from data. We separate taint information from the corresponding data and store the taints as a packed array in virtual memory. This organization still provides fast taint lookups, but makes the system memory and buses taint-agnostic; they remain the same regardless of whether tainting is used or not. To avoid extensive modifications to the processor core, FlexiTaint also separates taint and data processing by implementing taint-related logic as an in-order addition to the back-end of the pipeline. This leaves the processor core largely unmodified and allows data to be processed at full speed. Furthermore, by separating taint storage and processing from data, we avoid performance and storage overheads when tainting is not used. In this case, no memory needs to be allocated for taint storage and the processor core and caches operate on data exactly like they would without support for tainting.

With FlexiTaint, the maximum supported number of taint bits affects only the hardware of the back-end taint processing engine, so the hardware cost remains low even when supporting large taints. To allow programmability of taint propagation rules, FlexiTaint uses a Taint Propagation Cache (TPC) which is tagged by operation type (e.g. add, subtract, load, store, etc.) and input-operand taints. Misses in the TPC are handled by a user-level software handler, which computes the resulting taint for the missing operation-taints combination and inserts it into the TPC. To improve TPC performance, we employ a programmable filter that can be set up to bypass the TPC lookup for some instruction types and use simple common-case rules instead. This flexible approach allows FlexiTaint to act as an accelerator for a wide variety of taint propagation rules, which can be changed simply by changing the software of the TPC miss handler.

To evaluate our FlexiTaint mechanism, we program it with a set of "benchmark" tainting schemes. The first scheme is a representative of input-tainting schemes from prior work. The second scheme tracks which values in an application are valid heap pointers, which is useful for accelerating memory leak detection. This scheme is different from existing input tainting schemes, and we use it to show that FlexiTaint can be used to accelerate dynamic data flow analyses beyond traditional security uses. The third scheme we use in our evaluation is to simultaneously do input tainting and heap-pointer tracking, to evaluate the performance impact of multi-bit taints. Even for this scheme, our results indicate that FlexiTaint incurs low performance overheads (relative to a system without any taint propagation). These overheads are less than 1% on average and 8.4% worst-case for SPEC benchmarks, and 3.7% on average and 8.7% worst case on Splash-2 benchmarks.

2. Related Work

Static taint analysis was proposed to find format string vulnerabilities in C programs [20] or to identify potentially sensitive data [12]. Taint propagation is also similar to runtime type checking, where each object is "tainted" with its type and operations are checked for type-safe behavior in languages such as Java or CCured [9].

Perl [11] taints external data, and its taint propagation is compiled into the code by the just-in-time compiler or performed by the interpreter. Newsome et al. [10] use runtime binary rewriting to taint external inputs and propagate taints. Xu et al. [23] *tag* each byte of data, with elaborate policies to track these tags for security.

Hardware support has been proposed to improve performance of tainting and to accommodate self-modifying code and multithreading. Suh et al. [18] propose a low-overhead architectural mechanism that protects programs by tainting data from untrusted I/O and then propagating this taint. It provides some flexibility for particular taint propagation rules. Chen et al. [2] use the notion of pointer taintedness to raise alarms whenever a tainted pointer is dereferenced by the program. Minos [4] extends the microarchitecture with integrity bits and propagation logic that prevents control flow hijacking. TaintBochs [3] taints sensitive data and propagates this taint across system, language, and application boundaries. TaintBochs provides limited configurability to support different tradeoffs between security and the number of false alarms. The RIFLE architecture [21] supports runtime information flow tracking, and allows to enforce their own policies on their programs.

To our knowledge, Raksha [8] is the most configurable hardware taint propagation mechanism proposed to date. It supports multi-bit taints and has taint propagation registers that can be programmed to implement up to four different policies. Raksha also provides some flexibility in how the taints are propagated for each type of instructions. However, this flexibility is limited mainly to selecting whether or not a given input operand's taint should or shouldn't be propagated to the result, and to selecting whether the taint operands should be OR-ed or AND-ed. This is sufficient to efficiently implement most variants of existing tainting policies. In contrast, FlexiTaint can be used with *any* set of propagation rules in which the taint of the result depends only on the opcode and the taints of the operands.

Another important consideration for taint propagation schemes is how taints are stored and manipulated. Existing hardware tainting mechanisms tightly couple the data value and its taint: memory locations are extended with extra bits for the taint, and buses and caches are similarly affected. Unlike prior hardware support for taint propagation, FlexiTaint stores and processes taints separately from data. For storage of memory taints, FlexiTaint uses the approach used to store memory state (tags) in MemTracker [22]. Taints are stored as a packed array in virtual memory, allowing use of standard memory modules and existing OS memory management mechanisms. Taint processing in FlexiTaint is implemented as an in-order addition to the back-end of the processor pipeline to minimize impact on the already complex out-of-order core.

3. Overview of FlexiTaint

3.1. Storing taint information

Taint information must be associated with every word in memory. Previously proposed hardware support for taint propagation stores the taint along with the corresponding data, effectively widening the memory word. This approach has a number of drawbacks. First, it requires non-standard memory modules to keep the extra taint bits with each word. Second, these taint bits are wasted when no tainting is needed. Finally, special hardware and instructions are needed to access these taint bits, which makes bulk-manipulation (e.g. initialization) of taints difficult.

Previous hardware support for tainting also widens each cache block to accommodate the taint bits. Such widening makes the cache larger, more power-hungry, and possibly slower even when tainting is not used.

For FlexiTaint, our taint storage approach parallels the approach used in MemTracker [22] and HeapMon [15, 16] for their memory state. In particular, taints for data in an address space are kept as a packed array in a protected area within that address space. Given an address of a memory location, the corresponding taint can be found by simply indexing into this array. This organization allows us to use existing standard memory modules, buses, and caches. Taints can be kept protected from ordinary load/store accesses using existing page access permissions, but the system and library software can temporarily change these permissions to directly access taints for bulk-manipulation (e.g. to initialize them).

The only dedicated storage for taints in FlexiTaint is a separate small (4KBytes in our experiments) L1 cache. This cache provides bandwidth for taint accesses. The alternative would be to store taints in the existing L1 cache and add ports to it. However, this would make the L1 cache significantly larger and slower, affecting its hit latency (even when tainting is not needed).

Seemingly, a drawback of our taint storage approach is that taints now compete with data for space in secondary caches and below. However, our results (Section 6) show that this causes low performance overheads relative to a system with no tainting. If extra area is available, we believe it is better spent increasing the total capacity of the cache (to improve performance with or without tainting), instead of widening each cache block to provide dedicated taint bits.

A final advantage of decoupled taint storage is low-cost support for larger taints. In existing approaches, memory widening must accommodate the largest allowed taint, and those extra bits prove unnecessary if fewer taint bits are actually needed. With FlexiTaint, the packed taint array occupies only as much memory space as needed.

3.2. Processing taint information

Figure 1 shows how taint processing is integrated into the processor's pipeline in previously proposed schemes where taint information (dotted lines) must flow along with and be processed simultaneously with the data (full lines). Shaded areas in the figure indicate structures that are added or significantly changed to support tainting.

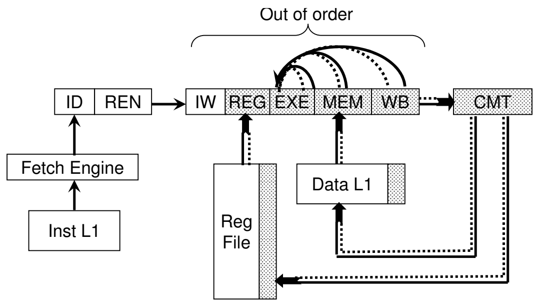

Figure 1. Previous tainting support

Much of the processor's logic and wire complexity involves moving, manipulating, or storing data values. To accommodate the taint along with the data, all these structures must be modified. These ubiquitous changes to the processor core require a tremendous re-design effort and make nearly every part of the core larger and longer-latency. It is unlikely that processor manufacturers will undertake such re-design solely to provide efficient tainting support.

In light of these considerations, we follow a different approach and implement the entire FlexiTaint taint processing accelerator as an in-order addition to the back end of the pipeline, as shown in Figure 2. This strategy has already been used for runtime verification [1] and memory checking [22], and has the advantage of keeping the performance-

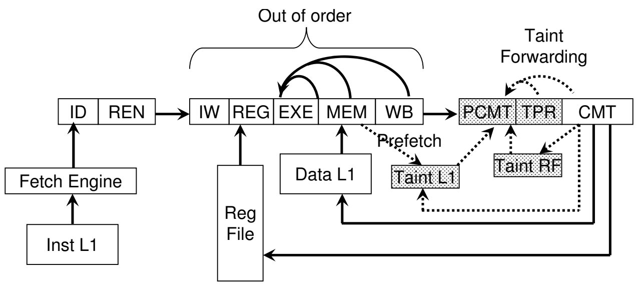

Figure 2. Processor pipeline with FlexiTaint

critical out-of-order dataflow engine largely unmodified. The added taint processing engine is easily turned off and bypassed when it is not needed.

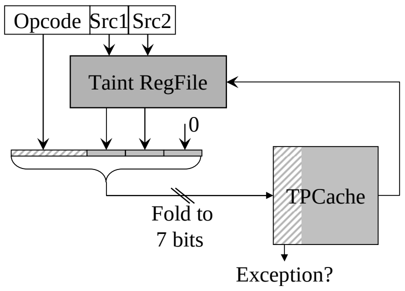

Figure 3. Taint propagation in FlexiTaint

We change the commit stage of the processor to pass instructions on to the FlexiTaint engine. We call this new stage pre-commit. The first FlexiTaint stage reads the taints of the register operands from the Taint Register File (TRF) (Figure 3). For load and store instructions, the taint of the memory operand is loaded from the TL1 cache. The next stage looks up the Filter Taint Propagation Table (Filter TPT) to determine if the taint propagation can be done using simple rules. If needed, a TPC lookup is performed. In case of a TPC miss, a software handler is invoked to determine the output taint for the given operation and the input taints. Next, the register result taint is written back to the TRF. After that, the instruction is ready to commit. More details are given in Section 4.

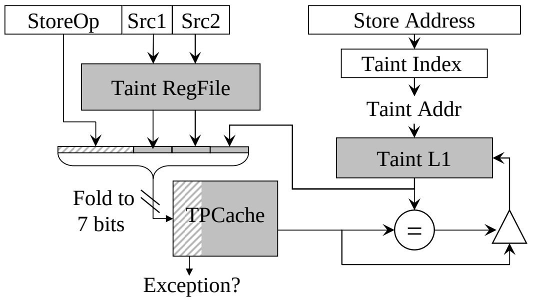

Figure 4. Taint propagation for stores

The commit for store instructions (Figure 4) involves the normal write to the data cache, and a write to the taint cache to update the taint of the destination memory location.

There are several advantages and efficiencies with this new approach to hardware taint propagation. First, the short in-order taint processing pipeline greatly simplifies the taint forwarding logic for both register and memory taints. Second, there is no register renaming, so the TRF only needs to store the taints for architectural registers. Third, support for multiple taint bits only affects the taint processing engine, where the TRF and the forwarding logic are much simpler and smaller than in the main processor core.

This approach also has several possible disadvantages. The main disadvantage is that the latency of misses in the TL1 cache is fully exposed because they stall the in-order taint processing pipeline. Fortunately, taint addresses can easily be computed from data addresses, so we issue a taint prefetch as soon as the data address is available. Because the taint access pattern is a more compact version of the data access pattern, a TL1 prefetch miss is often overlapped and its latency hidden by a DL1 miss. As a result, when a load instruction comes to the pre-commit stage and requests its taint from the TL1 cache, the taint prefetch is usually already complete and the access is a TL1 hit.

The second disadvantage of our back-end taint processing approach is that it could delay instruction commit by several cycles, increasing the pressure on the ROB, physical registers, and other processor resources. However, with the use of structures like TPC and a Filter TPT, we find that this delay is short and in our experiments it has a modest effect on performance.

A third disadvantage of our scheme is that the decoupled approach to data and taint storage can result in consistency problems in multiprocessors. More specifically, a data write and its corresponding taint write must happen atomically to prevent a read from getting the new data with the old taint or vice versa. Similarly, a data read and a taint read must also happen atomically. We found that read atomicity can be provided by leveraging existing load-load replay mechanisms. We also find that most taint writes are silent writes [6] that can easily be eliminated. For non-silent taint writes, we ensure atomicity by making sure that both writes (taint and data) are L1 cache hits before allowing either of them to modify its cache block.

Finally, a fourth potential issue in our FlexiTaint engine is that dependences between instructions may cause in-order stalls and create a performance bottleneck. Fortunately, our FlexiTaint engine has two major advantages over ordinary in-order processors. First, in-order processors suffer stalls when they encounter an instruction that depends on a long-latency instruction. In FlexiTaint, all instructions have similar short-latency taint lookups, and cache misses are largely eliminated by prefetching. The second advantage is that most taint propagation operations are simply copying input taints to the destination, which allows us to eliminate most dependences between same-cycle taint propagation operations (see Section 3.5).

3.3. Programmable Taint Propagation

Because it is an accelerator that should be able to implement many different tainting policies, FlexiTaint allows software to compute the resulting taint for a given combination of the instruction opcode and input taints. However, software intervention for every instruction would introduce huge overheads. To avoid these overheads, FlexiTaint uses a Taint Propagation Cache (TPC) to memoize resulting taints for recently seen combinations of opcode and input taints.

The TPC is indexed by concatenating the opcode (or opcode class) and the taints of all source operands (Figure 3). Each entry in the TPC contains the resulting taint, and also a bit that indicates whether an exception should be raised when this combination of opcode and input taints is encountered. If the TPC lookup results in a TPC miss, a software handler is invoked to compute the resulting taint for that combination of opcode and input taints. This is then inserted into the TPC similar to how TLB entries are filled by software TLB miss handlers. This approach allows us to specify a taint propagation policy simply by writing a TPC miss handler. The address of this handler is kept in a special CPU control register (TPC Handler Register).

In our experiments, we use a small (128-entry) on-chip TPC, which is directly mapped to allow single-cycle lookups. To avoid caching stale TPC entries, the TPC is flash-cleared whenever the TPC Handler Register is changed. This allows us to change the taint propagation policy (e.g. on a context switch) by just writing a new handler's address into the TPC Handler Register. Note that tainting is often used to detect attacks, so attackers must not be allowed to change the TPC Handler Register or the code of the handler itself. For this reason, the TPC Handler Register can only be modified in kernel mode.

3.4. Taint Manipulation Instructions

In most taint propagation schemes, there are high-level events that affect taint propagation but are not readily recognizable at the hardware level. For example, an input tainting scheme typically untaints data that has been range-checked to avoids false alarms for jump-table implementations. However, in most ISAs a range-check involves a sequence of instructions that are difficult to recognize by the hardware as a range-check.

To allow high-level events to be conveyed to our FlexiTaint hardware, we add a small number of new `taintr` and `taintm` opcodes. These instructions are treated as no-ops in the main processor pipeline, but are processed in the FlexiTaint engine. A `taintr` instruction has a register-to-register format, and based on the opcode and the the input taint values, FlexiTaint performs a TPC lookup to determine the new taint of the destination register. This instruction allows us to change the taint associated with a register value, without changing the value itself. Similarly, a `taintm` instruction has the format of a store operation, but only affects the taint of the memory location. The system software can use these `taintr` and `taintm` instructions to mark high-level events and set the propagation rules for these instructions to achieve the needed taint propagation actions. For example, a `taintm` instruction can be added to the *message receive* code to taint the input data, and `taintr` can be used to, for example, untaint a value that has been range-checked. Note that the mapping between high-level events and these new opcodes is determined by the programmer or the system, not by the ISA or hardware itself. To indicate a specific high-level event, we choose an unused opcode, put it in the code to signal the event, and change the TPC miss handler to perform correct tainting for the event when that opcode is encountered.

Instead of using new opcodes, high-level events could be indicated by trapping into the system, which can then directly read/write the taint array in memory to implement the needed tainting behavior. However, such traps may add significant overheads when high-level of interest are frequent.

Finally, we note that any hardware taint propagation scheme requires changes to the system and libraries to indicate high-level events that cannot be identified by hardware alone. Our implementation of FlexiTaint differs only in that it provides a generic set of ISA extensions for this purpose. This is consistent with FlexiTaint's role as an accelerator for taint propagation - it speeds up the time-consuming activity of instruction-to-instruction taint propagation, and relies on existing taint propagation schemes for system-level support and for specific sets of rules it is programmed with.

3.5. Fast Common-Case Taint Propagation

Although the TPC is small, if it is accessed for every instruction it would need to be multi-ported to keep up with the throughput of the multiple-issue processor core. However, multi-porting could slow the TPC down, make it power-hungry, and make future enhancements or upgrades to larger TPCs costly.

To reduce the number of TPC accesses and keep the TPC single-ported, we rely on two key observations that hold for most (but not all) types of instructions in the schemes we studied. First, if inputs are untainted (zero-taint), the output is also untainted. Second, if only one of the operands has a non-zero taint, the result taint is simply a copy of the non-zero input taint.

To exploit these observations, we use a programmable *Filter Taint Propagation Table* (Filter TPT) to selectively enable these optimizations for opcodes to which they are applicable according to the current set of taint propagation rules. The Filter TPT is indexed only by opcode, and each entry has only two bits that tell us which common-case optimizations can be used (Table 1). With 256 opcodes, the Filter TPT is a simple 512-bit table (no tag checks). Although it has multiple read ports to support separate lookups for each instruction issued in a cycle, this small table uses little on-chip area and is fast enough for single-cycle lookups.

Value	Meaning
00	Use Taint Propagation Cache (TPC) for this opcode.
01	If all source taints are zeros, destination taint is zero. Otherwise, like **00** (use TPC).
10	If only one non-zero source taint, copy it to destination taint. Otherwise, like **01**.

Table 1. Meaning of Filter TPT entries

We note that these optimizations are based on observations that were made in our two example taint propagation schemes (Section 5). It is possible to devise a set of taint propagation rules for which TPC lookups are always needed (all Filter TPT entries are **00**). Fortunately, now and in the foreseeable future the most common use taint propagation is tainting of input-derived data to detect security violations. These schemes are similar to the first example tainting schemes used in our experiments, so we expect such schemes to show similar benefit from the Filter TPT.

Another benefit of using our Filter TPT is that its optimizations also allow us to eliminate most of the dependences between same-cycle instructions. If the TPC lookup is not needed, note that the resulting taint is either zero or equal to the taint of one of the operands. In such cases, the "computation" of the taint is trivial and the source taint can be directly forwarded in the same cycle to a dependent instruction. A similar idea (but with a different implementation) was used for elimination of move instructions in RENO [13]. In FlexiTaint, we have the added advantage that most of the taint propagation operations are taint moves, even when the corresponding data operation for the same instruction is not a move.

4. FlexiTaint Implementation

With FlexiTaint, the front-end and the out-of-order dataflow engine of the processor core are largely unmodified. The most significant modification to these parts of the processor is that load and store instructions also compute the taint address and issue a non-binding taint prefetch into the TL1.

The main FlexiTaint pipeline (Figure 2) begins when the instruction is otherwise ready to commit. As a result, FlexiTaint receives instructions in-order, does not receive any wrong-path or otherwise speculative instructions, and gets already-decoded instructions. This greatly simplifies the implementation of our FlexiTaint engine.

FlexiTaint starts off by fetching source taints, (in parallel) looking up the Filter TPT, and (also in parallel) checking dependences. The next step is to check which source taints are zero and whether the value found in the Filter TPT allows us to use a common-case optimization. If one of these optimizations can be used, the taint propagation is trivial - the resulting taint is either zero (if all source taints are zero and the Filter TPT entry has a value other than 00) or equal to the non-zero source taint (if there is only one non-zero source taint and the Filter TPT entry is 10). If the Filter TPT and the source taints are such that a TPC lookup is needed, the next step is to look up the TPC entry using the opcode and the source taints as an index and tag. In this case, a same-cycle dependent instruction (and all subsequent instructions) is stalled until the next cycle. If no TPC lookup is needed, same-cycle forwarding of the original source taint (see Section 3.5) is used to avoid in-order stalls. Next, the resulting taint is written to the Taint Register File (TRF) if the instruction's destination is a register. Finally, the instruction is ready to commit. In our current implementation, the FlexiTaint pipeline has a total of four stages in addition to the regular pipeline, two to look up the Filter TPT, TL1, and register taints, one stage for actual taint propagation (TPC lookup or trivial propagation with same-cycle forwarding), and one to finally commit.

Another consideration is handling of TPC misses. A TPC miss is an exception that triggers execution of a software handler. We find that such exceptions are very rare in our experiments due to a combination of several factors. First, most dynamic instructions are amenable to common-case optimizations and do not access the TPC at all. Second, for instructions that do access the TPC, there is significant locality in the values of source taints and opcodes (some opcodes are used much more frequently than others and some taint values are much more common than others).

4.1. Multiprocessor Consistency Issues

Outside the processor and L1 caches, FlexiTaint memory taint values are stored like any other data, so they are automatically kept coherent in a multiprocessor system. However, FlexiTaint does raise a few issues with respect to consistency. In particular, sequential consistency and several other consistency models assume that a load or a store instruction appears to execute atomically. For brevity, we only discuss sequential consistency, but our discussion can easily be extended to other models.

The main problem is that FlexiTaint stores taints in memory separately from the corresponding data. As a result, a load (which reads the data and its taint) on one processor and a store (which writes both data and its taint) on another processor may be executed such that, for example, the load obtains the new data but the old taint.

To prevent this inconsistency, data and the taint must be read atomically in a load and written atomically in a store. Load (read) atomicity must ensure that the data value is not changed between the time data is read in the main processor core and the time the taint is read in the FlexiTaint pipeline. Existing replay traps can be leveraged to accomplish this by treating data loads as "vulnerable" to invalidation-caused replay until the corresponding taint is read. We implemented this behavior, and in our experiments we observe practically no performance impact due to such replays because they are exceedingly rare.

Write atomicity for a store instruction is more challenging. To prevent speculative writes to the L1 cache, modern processors delay cache writes until the instruction can commit. When FlexiTaint is active, we also delay the taint write until the instruction commits. As a result, both writes (taint and data) need to be performed atomically at commit time. To simplify the implementation, we change the commit logic to only perform both writes if both are hits - if either data or the taint write is a miss, we do not allow the other to modify its cache. To accomplish this, we check the hit status of both accesses after tag checks, and suppress the actual write in one cache if the other indicates a miss. Once the miss is serviced, both writes are re-tried. There are several factors that prevent this from having a significant effect on performance. First, a TL1 access is nearly always a hit (due to prefetches), so DL1 write hits are rarely delayed by TL1 misses. Second, a tag check in the small TL1 are completed well before the DL1 tag check, so the propagation of the TL1 hit signal does not delay a DL1 hit. Finally, many taint writes are silent writes [6]. Because FlexiTaint already reads the memory location's taint for store instructions (Figure 4), we compare the new taint with the old one and only write the new taint if it differs from the old one. This optimization allows most store instructions to write only data, and also reduces the coherence traffic on memory blocks that store taints because they become dirty less often.

4.2. Initialization and OS Interaction

With FlexiTaint, the processor context of a process is extended to include the TPC Handler Register (TPCHR), the FlexiTaint Configuration Register (FTCR), the Memory Taint Base Register (MTBR), and the Filter TPT content. The TPCHR contains the address of the TPC miss handler, and is discussed in Section 3.3. The FTCR contains the taint size, which can be from zero to sixteen bits in our current implementation. Taint size of zero bits indicates that tainting is not used, and it turns off taint propagation circuitry and the TL1 cache. The MTBR contains the virtual address of the packed array that stores taints of memory locations. The Filter TPT is already discussed in Section 3.5.

To initialize FlexiTaint, we allocate the memory taint array, initialize it, and protect it from ordinary user-level accesses so only FlexiTaint-initiated accesses can modify these taints. Next, we load the address of the TPC miss handler into the TPC Handler Register. We then load the filtering rules into the Filter TPT and the address of the memory taint array into the Memory Taint Base Register. Finally, we write the number of taint bits to the FlexiTaint configuration register, which enables the FlexiTaint mechanism.

For context switching, we simply save/restore the three registers (TPC Handler, Memory Taint Base, and FlexiTaint Configuration) and the Filter TPT content to/from the context of the process. This save/restore is very fast: only three additional registers are saved/restored, and the Filter TPT is very small. In our implementation, it is only 64 bytes (512 bits) in size.

As our memory taints have their own virtual and physical addresses, they are also fully compatible with OS mechanisms such as copy-on-write optimizations on process forking, paging and virtual memory, disk swapping, etc.

5. Evaluation Setup

To evaluate our FlexiTaint accelerator, we use two example taint propagation schemes. The first is input tainting similar to dynamic information flow tracking [18]. This scheme is intended to be representative of tainting schemes that look for security violations by tainting input-derived data and detecting when such data is used in insecure ways (e.g. as data pointers, or jump addresses). The second example scheme "taints" values that are valid heap pointers. This can be used to speed up pointer identification for memory leak detection. This scheme's propagation rules are very different from those of input-tainting, so it also illustrates how easily FlexiTaint can be used with different taint propagation policies. We note that this evaluation is not intended to show that FlexiTaint can detect a specific attack or a class of attacks. FlexiTaint can be programmed to follow a wide variety of taint propagation schemes, and the attack detection comes from these schemes. Consequently, we do not claim taint propagation schemes in Tables 2 and 3 as our contributions. They are merely example schemes to show how FlexiTaint can be used and to evaluate its impact on performance.

Instruction	Input-taint propagation rule
ALU-Op R1,R2,R3 (add, sub, etc.)	Taint(R1) = Taint(R2) OR Taint(R3).
mov R1,R2	Taint(R1)=Taint(R2).
ld R1,offset(R2)	Taint(R1)= Taint(R2) OR Taint(Mem[R2+offset]).
st offset(R1),R2	Taint(Mem[R1+offset])= Taint(R1) OR Taint(R2).
taintm0 offset(R1)	Taint(Mem[R1+offset])= INPUT-TAINT.
Jump R1 (branch, jump, etc.)	If Taint(R1)= INPUT-TAINT, raise exception.

Table 2. External Input Tracking.

Instruction	Pointer-taint propagation rule
add R1,R2,R3	Taint(R1)=Taint(R2) OR Taint(R3). Exception if Taint(R1) AND Taint(R2)
sub R1,R2,R3	Taint(R1)=Taint(R2) XOR Taint(R3).
mov R1,R2	Taint(R1)=Taint(R2).
ld R1,offs(R2)	Taint(R1)= Taint(Mem[R2+offs]).
st offs(R1),R2	Taint(Mem[R1+offs])= Taint(R2).
taintr0 R1	Taint(R1)=POINTER-TAINT.

Table 3. Heap Pointer Tracking.

5.1. Taint Propagation Schemes

Table 2 shows the rules we use for input tainting. In this table, `taintm0` is the first of our `taintm` opcodes, which we added to input/output libraries to indicate memory into which external inputs were just received in `read` and similar input functions.

These rules can easily be converted into TPC and Filter TPT entries. For example, according to the table, the TPC entry for a *store* instruction with both the address register and value register tainted would be to raise no exception and to taint the target memory location. Also, the same rule allows us to set the Filter TPT entry for `store` to ”10”, avoiding TPC lookups for a `store` with input taint combinations other than the one just described.

Our simulator uses the MIPS instruction set, where R0 is hard-wired to the value of zero and `add rX,rY,R0` is used to move values from `rY` to `rX`. In keeping with this, we also hard-wire the taint for register R0 to zero (no taint), so no separate rule for *mov* is needed. Alternatively, uses of `add` as `mov`, zero-out uses of `xor` and `and`, etc. can be decoded as separate opcodes that have their own taint propagation rules.

For heap pointer tracking, we use rules shown in Table 3. The `taintr0` is the first of our `taintr` opcode, which we added to memory allocation libraries to indicate return values of heap allocation functions such as `malloc`.

Note that these propagation rules are different from those in Table 2. For example, the `store` instruction only propagates the pointer-taint of the value register, but ignores the heap-pointerness of the address register. As a result, we can only set the Filter TPT entry for `store` to “01” (see Table 1).

For propagating both taints, we use a two-bit taint where the first bit is propagated according to Table 2 (input tainting) and second bit is propagated according to Table 3 (pointer tainting). For example, the TPC entry for a `store` instruction with the address register taint of “01” (heap pointer) and value register taint of “10” (input-derived value), the exception bit would not be set and the taint for the result (memory location) would be “10” because the input taint of the value register is propagated but the pointer taint of the address register is not. Note that the Filter TPT entry for `store` can only be set to “01” to eliminate TPC lookups when both source registers are untainted.

5.2. Benchmark Applications

We use all applications from the SPEC CPU 2000 [17] benchmark suite. For each application, we use the reference input set in which we fast-forward through the first 10% of the execution to skip initialization, and then simulate the next one billion instructions in detail. We note that our fast-forwarding must still model all taint creation and propagation to provide correct taint state for the simulation. In order to evaluate the multi-threaded workloads, we simulate benchmarks from the Splash2 [19] suite (no fast-forwarding).

5.3. Simulator and Configuration

We use SESC [14], an open-source execution-driven simulator, to simulate an 8-core system with Core2-like, four-issue out-of-order superscalar cores running at 2.93GHz. Data L1 caches are 32KBytes in size, 8-way set-associative, dual-ported, with 64-byte blocks. The shared on-chip L2 cache is 4MBytes in size, 16-way set-associative, single-ported, with 64-byte blocks. The processor-memory bus is 64 bits wide and operating at 1333 MHz. In FlexiTaint configurations, taint L1 caches are 4KBytes in size, 4-way set-associative, dual-ported, and also with 64-byte blocks.

6. Evaluation

We conduct experiments to evaluate the performance of FlexiTaint when it is programmed to implement input tainting, pointer tainting, and also when it is programmed to simultaneously implement both schemes. Figure 5 shows the execution time overhead for all SPEC2000 and Splash-2

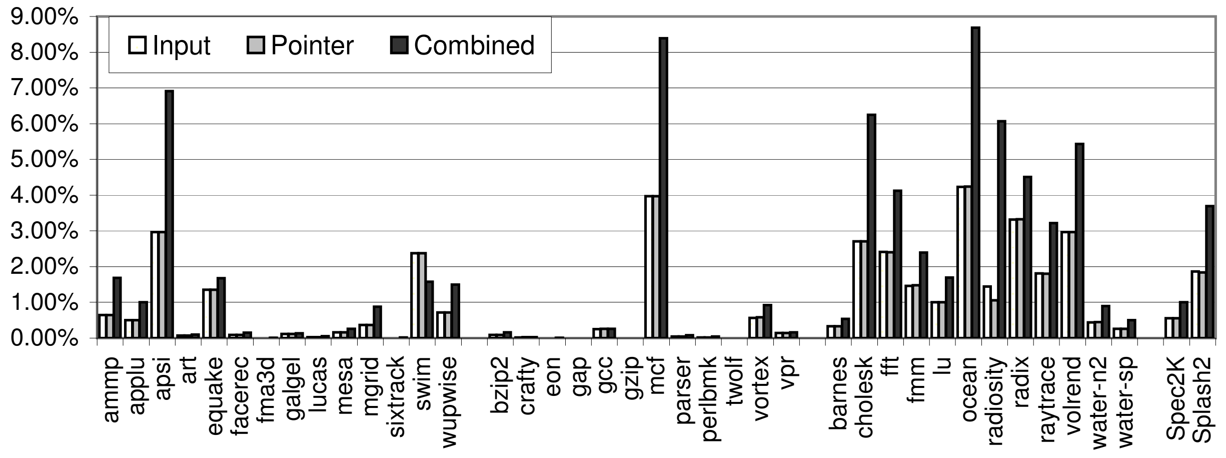

Figure 5. Performance overhead of taint propagation with FlexiTaint.

applications. For SPEC2000, we observe worst-case overheads of 8.4% (in mcf) and average overheads of about 1%, even for the combined taint propagation scheme. In benchmarks with above-average performance overheads, most of the overhead is caused by increased L2 miss rates and increased L2 port contention because the L2 cache is used for both data and taints. The L2 capacity problem is also dominant in most applications, which explains the small differences in overheads between one-bit and two-bit schemes. In swim, the overhead in the combined two-bit scheme is slightly lower than for one-bit schemes. As different memory locations are accessed for taints when taint sizes are different, the overlap of cache misses with other operations in these executions is also different. In swim, the combined scheme suffers more cache misses (as expected), but in single-bit schemes there is less overlap and, as a result, they suffer more overhead.

For Splash-2 benchmarks, we observe modest performance overheads - about 3.7% on average and 8.7% worst-case (in ocean). Both input and pointer taint propagation schemes show similar overheads for all benchmarks except radiosity where the input scheme has slightly higher overhead than the pointer scheme. This is due to increased number of taints propagated between memory and registers in the input tainting compared to the pointer scheme.

Figure 6 shows a breakdown of dynamic instructions according to which Filter TPT optimizations are applied (data shown is for the 2-bit combined tainting scheme). Dynamic instructions that could not benefit from Filter TPT common-case optimizations are further classified into those that hit and those that miss in the TPC. Note that the instructions shown as "No Taint" are not all the instructions whose operands carry no taint. Instead, these are instructions whose operands carry no taint *and* the Filter TPT allows that opcode to use the "no taint yields no taint" common-case optimization without accessing the TPC. Similarly, "One Source Taint" instructions in Figure 6 are those that have one operand tainted with a non-zero taint *and* the Filter TPT allows the instruction's opcode to use the common-case optimization of copying the non-zero source taint to the destination taint without using the TPC. We find that most of the instructions processed by the FlexiTaint engine can be handled by one of these common-case rules. Instructions that access the Taint Propagation Cache (TPC) are infrequent enough, and represent only up to 1.9% (in *mcf*) of all dynamic instructions. As expected, TPC misses are extremely rare.

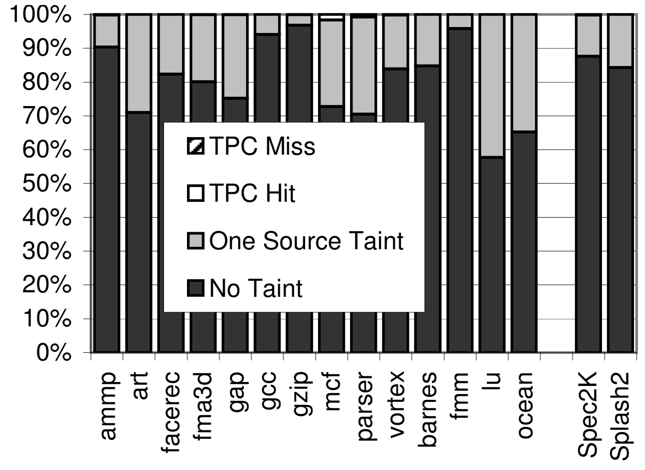

Figure 6. Use of FlexiTaint optimizations.

These results indicate that our combination of TPC handlers for programmability, TPC for memoization of frequently used rules, and Filter TPT for common-case optimizations can achieve a very high level of programmability with low performance overheads. Also, the TPC can indeed be small and does not need to be multi-ported because most instructions do not actually access it. However, in all of the applications the TPC *is* accessed a non-trivial number of times, which indicates that the TPC and its software miss handler are still needed to support less common taint propagation cases that are specific to each set of taint propagation rules.

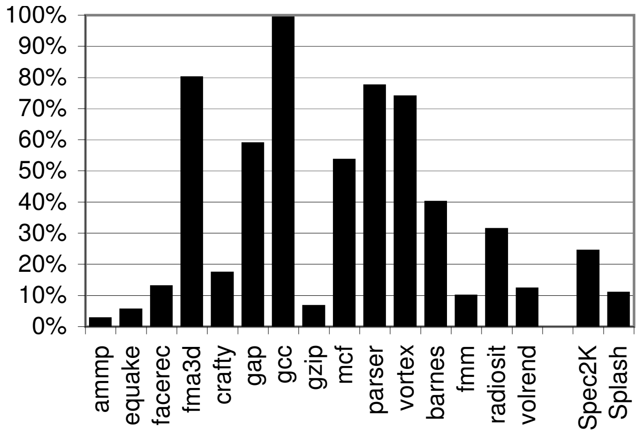

Figure 7. Non-silent taint writes.

Figure 7 shows the number of non-silent memory taint writes as a fraction of the number of dynamic instances of store instructions processed by the FlexiTaint engine. We only show a subset of applications to illustrate the range of the fraction. The averages for SPEC2000 and Splash-2 shown in the figure cover all of the applications. Because every store instruction produces a resulting taint, without detection of silent taint writes, all store instructions would cause FlexiTaint to write the resulting taint to the TL1 cache. From this figure, we observe that in many benchmarks nearly 80% of store instructions do not change the taint of the target memory location. On the other hand, in benchmarks like gcc nearly 99% of stores have non-silent taint writes.

6.1. Sensitivity Analysis

The TL1 cache is the largest on-chip structure we added to support FlexiTaint, and we used 4KByte TL1 caches in our experiments. However, we performed additional experiments with 2KByte and 8KByte TL1 caches to determine how the size of the TL1 affect the performance of FlexiTaint. These experiments indicate that a larger 8KB TL1 results in no noticeable performance improvement over our default 4KB TL1 cache. For the smaller 2KByte TL1 cache, we find that the performance overhead increases to 2.8% on average and nearly 14% worst-case (in *mcf*). We conclude that our default 4KByte cache is well-chosen for one- and two-bit taints, but a scheme that uses four-bit taints may need a larger (e.g. 8KByte) cache to avoid some increase in overheads due to TL1 contention.

Our results in Figure 5 show that multi-threaded (Splash-2) applications have higher overheads than single-threaded (SPEC) ones. This is largely due to false sharing of taint blocks. A single taint block corresponds to numerous data blocks: with 2-bit taints, a single taint block corresponds to 16 data blocks. When two processors access different data blocks, the taints for those data blocks can be in the same taint block, causing false sharing.

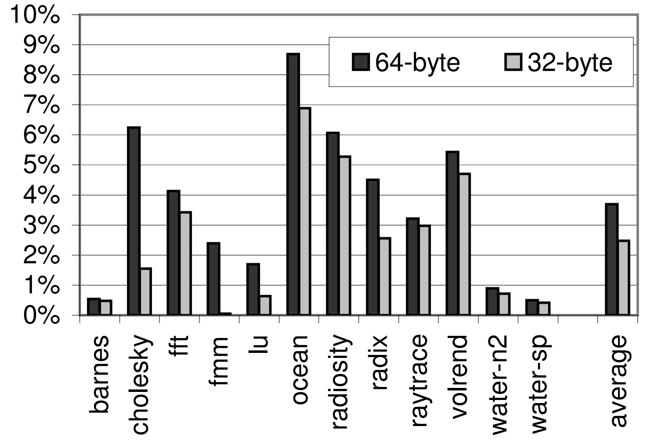

Figure 8. Effect of TL1 line size in Splash-2.

There are several well-known ways to reduce false sharing. One would be to pad and align data structures. Several Splash-2 applications already use padding to reduce false sharing on data blocks, but the granularity of this padding is insufficient for taint blocks. If the L2 cache is shared, false sharing of taints can be reduced using a smaller block size in the L1 taint cache. Figure 8 compares performance of FlexiTaint with 64-byte and 32-byte blocks in the taint cache (DL1 and L2 use 64-byte blocks in both configurations. We observe performance improvements with smaller taint block sizes, and in several applications this improvement is dramatic. This confirms that false sharing is indeed present, and also shows that it can be reduced without affecting the design of existing DL1 and L2 caches.

With private L2 caches, coherence actions occur between L2 caches. Even if the block size in the TL1 cache is smaller, a TL1 miss still results in a coherence request for an entire larger L2 block. A possible solution would be to use a sectored L2 cache. A data L1 miss would then fetch an entire block into the L2 cache, but a taint L1 miss would only fetch a particular sub-block (sector). We leave this and other more sophisticated schemes for future work.

6.2. Effect of limited programmability

To demonstrate the benefits of FlexiTaint's full programmability, we artificially limit its programmability to only handle propagation rules that can also be handled by previously proposed hardware support, then use the resulting accelerator on the heap-pointer tracking scheme. Most of the rules for heap-pointer tracking (Table 3) can be handled by this limited scheme. However, rules for `add` and `sub` cannot. In all input tainting schemes, both of these opcodes simply perform a logical OR of input taints to produce the propagate the output taint, without raising any exceptions. In heap-pointer tracking, addition or subtraction of two heap pointers produces a non-pointer, and addition of two pointers is meaningless and indicates a probable bug. As a result, the heap-pointer tracking rule for `add` is to propagate the taint if only one source operand is pointer-tainted, but to raise an exception (invoking a software handler to record a possible bug) when both operands are pointer-tainted. The heap-pointer tracking rule for `sub` is to propagate the taint if only one source operand is tainted, but produce an untainted result if both source operands are tainted. Effectively, the taint propagation rule for `sub` is a logical XOR of input taints. A limited-programmability scheme must raise an exception for each `add` and `sub` instruction and implement correct taint propagation for these instructions in an exception handler. In our evaluation, we do not actually implement this exception handler. Instead, we model its overhead by adding a (rather optimistic) 5-cycle penalty each time an `add` or `sub` is encountered.

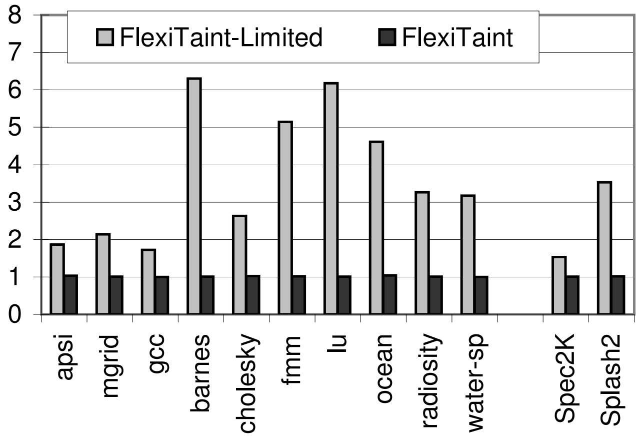

Figure 9. Effect of limited programmability.

The result of this evaluation is shown in Figure 9, which shows execution times with a limited-programmability scheme and with our full FlexiTaint mechanism. Execution times are normalized to a baseline that does no taint propagation. These results indicate that limited-programmability hardware support incurs large performance overheads when it is used on schemes that are not sufficiently amenable to the particular programmability limitations.

Note that we do not claim that other hardware schemes cannot be extended to support the particular taint propagation rules needed to support `add` and `sub` for heap-pointer tracking. For example, the original description of Raksha [8] includes no support for XOR-in input taints, but such support can easily be added.

We argue that the needs of future taint propagation schemes are difficult to predict at hardware design time, and we claim that by providing more programmability we run far less risk of being unable to efficiently handle those future schemes. In other words, any existing hardware mechanism can be extended with a setting that allows it to support a particular tainting rule it currently does not support, but this obsoletes systems that have the previous iteration of the hardware support. Our approach is to avoid this by designing our FlexiTaint accelerator to be highly programmable to begin with.

6.3. Validation

As mentioned previously, input-tainting itself is not our contribution, and we use heap-pointer tracking mostly as an example of a scheme with propagation rules different from those in input tainting. Our FlexiTaint accelerator is intended to provide a programmable, low-cost, and implementable substrate that allows implementation of various taint propagation schemes with low performance overheads. As a result, our evaluation focuses on performance. However, we do verify that our implementation of input tainting and pointer tracking correctly tracks input-derived and pointer values throughout the program. We also validate heap pointer tracking by verifying that tainted variables are indeed heap pointers and that non-pointers remain untainted. We verify input tainting by injecting buffer overflows and verifying that they are indeed detected. Interestingly, some benchmarks (twolf and gcc) have a number of dynamic instructions that produce values tainted with both the pointer taint and the input taint. For such occurrences, we examine the source code of the application to confirm the correct behavior of our scheme.

7 Conclusions

This paper proposes and evaluates FlexiTaint, a programmable hardware accelerator for taint propagation. FlexiTaint is implemented without modifying the system bus or main memory, and without extensive modifications to the performance-critical front-end pipeline of the processor or to its out-of-order dataflow engine. FlexiTaint stores memory taints as a packed array in virtual memory, and its taint processing engine is implemented as an in-order addition to the back end of the CPU pipeline. FlexiTaint is also fully programmable, using a Taint Propagation Cache (TPC) backed by a software miss handler to determine the taint of an instruction's resulting data value, using as inputs the type of operation (its opcode) and the taints of the operands. A Filter TPT is also used to reduce the number of accesses to the TPC for common-case optimizations.

Our results indicate that FlexiTaint incurs very low performance overheads, even when using a two-bit taint that simultanously tracks two very different properties with different taint propagation rules. We also evaluate FlexiTaint on Splash-2 benchmarks and demonstrate that it operates correctly and with low overheads even in a multi-core system.

8. Acknowledgements

This work has been supported, in part, by NSF under grants CCF-0447783, CCF-0541080, and CCF-0347425, and by gifts from IBM. The authors would also like to thank the anonymous reviewers and Dr. David Kaeli for their valuable feedback, which was of great help in improving this paper.

References

[1] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. *Proc. 32nd International Symposium on Microarchitecture*, 1999.

[2] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. *Proc. 2005 International Conference on Dependable Systems and Networks*, 2005.

[3] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. *Proc. 13th USENIX Security Symposium*, 2004.

[4] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *Proc. 37th International Symposium on Microarchitecture*, 2004.

[5] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. *Proc. European Conference in Computer Systems*, 2006.

[6] K.Lepak and M.Lipasti. Temporally Silent Writes. *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[7] J. Kong, C. Zou, and H. Zhou. Improving Software Security via Runtime Instruction Level Taint Checking. *Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[8] M.Dalton, H.Kannan, and C.Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. *Proc. 34th International Symposium on Computer Architecture*, 2007.

[9] G. Necula, S.McPeak, and W.Weimer. CCured: typesafe retrofitting of legacy code. *Proc. 29th Symposium on Principles of Programming Languages*, 2002.

[10] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Proc. 12th Annual Network and Distributed System Security Symposium*, 2005.

[11] O'Reilly. *http://search.cpan.org/dist/perl/pod/perlsec.pod*. Perl Security, 2005.

[12] P.Broadwell, M.Harren, and N.Sastry. Scrash: A System for Generating Secure Crash Information. *Usenix Security Symposium*, 2003.

[13] V. Petric, T. Sha, and A. Roth. RENO: A Rename-Based Instruction Optimizer. *Proc. 32nd International Symposium on Computer Architecture*, 2005.

[14] J. Renau et al. Sesc. *http://sesc.sourceforge.net*, 2006.

[15] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: a Low Overhead, Automatic, and Programmable Memory Bug Detector. In *Proc. 1st IBM T.J. Watson Conference on Interaction between Architecture, Circuits, and Compilers (PAC2)*, 2004.

[16] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM Journal of Research and Development*, 50(2/3):261–275, 2006.

[17] SPEC. Standard performance evaluation corporation. *http://www.spec.org*, 2004.

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

[19] S.Woo, M.Ohara, E.Torrie, J.Singh, and A.Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. 22nd International Symposium on Computer Architecture*, 1995.

[20] U.Shankar, K.Talwar, J.Foster, and D.Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. *Usenix Security Symposium*, 2001.

[21] 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. *Proc. 37th International Symposium on Microarchitecture*, 2004.

[22] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. *Proc. 13th International Symposium on High Performance Computer Architecture*, 2007.

[23] W. Xu, S. Bhatkar, and R.Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. *Usenix Security Symposium*, 2006.