Flexible Hardware Acceleration for Instruction-Grain Program Monitoring

Shimin Chen[1], Michael Kozuch[1], Theodoros Strigkos[2], Babak Falsafi[3], Phillip B. Gibbons[1], Todd C. Mowry[1,2], Vijaya Ramachandran[4], Olatunji Ruwase[2], Michael Ryan[1], Evangelos Vlachos[2]

[1]*Intel Research Pittsburgh* [2]*Carnegie Mellon University*
[3]*École Polytechnique Fédérale de Lausanne* [4]*University of Texas at Austin*

Abstract

Instruction-grain program monitoring tools, which check and analyze executing programs at the granularity of individual instructions, are invaluable for quickly detecting bugs and security attacks and then limiting their damage (via containment and/or recovery). Unfortunately, their fine-grain nature implies very high monitoring overheads for software-only tools, which are typically based on dynamic binary instrumentation. Previous hardware proposals either focus on mechanisms that target specific bugs or address only the cost of binary instrumentation. In this paper, we propose a flexible hardware solution for accelerating a wide range of instruction-grain monitoring tools. By examining a number of diverse tools (for memory checking, security tracking, and data race detection), we identify three significant common sources of overheads and then propose three novel hardware techniques for addressing these overheads: Inheritance Tracking*,* Idempotent Filters*, and* Metadata-TLBs*. Together, these constitute a general-purpose hardware acceleration framework. Experimental results show our framework reduces overheads by 2–3X over the previous state-of-the-art, while supporting the needed flexibility.*

1 Introduction

Systems designers have traditionally focused on maximizing performance, and more recently on minimizing power. From a user's perspective, however, both of these issues are of secondary concern when the software is misbehaving. In other words, if the software is broken, it is little consolation that it is misbehaving quickly or power-efficiently. As systems have become faster over the years, the corresponding increases in both software and hardware complexity have raised concerns that applications and systems are becoming increasingly error-prone. While writing bug-free code has always been difficult, recent studies suggest that bug rates are getting worse over time as software complexity increases [4], despite the software industry's pre-release testing efforts. In a networked world, even obscure bugs—benign under normal conditions—can leave a system vulnerable to security attacks after the code has been released [31].

There is a long history of developing tools to help diagnose and fix software problems. These tools can be invoked at various phases of the software development and execution cycle: e.g., *static* tools attempt to identify problems before the program executes [2, 9, 11], *post-mortem* tools attempt to reconstruct what went wrong after the application crashes [15, 39, 40], and *dynamic* tools—which we call *lifeguards*—monitor an application as it executes to diagnose and hopefully either contain or fix problems [1, 10, 14, 20, 27]. These three classes of tools are generally complementary. Moreover, the granularity of software execution events that lifeguards care about form a spectrum, from system-call-level [12, 24] to instruction-level [18, 20, 21, 25, 27]. *Instruction-grain lifeguards*, which perform invariant checking and analysis at the granularity of individual instructions, have two unique advantages: (i) highly-detailed information regarding dynamic events, such as memory references, address computations, and information flow, is available at the instruction level, and (ii) software errors may be captured earlier and more accurately. The former enables a wide range of powerful lifeguards (e.g., detecting memory-access violations [18, 20], data races [27], and security exploits [21, 25]), while the latter provides a better starting point for damage containment, on-site diagnosis [32], and hopefully on-the-fly fixes and recovery [26].

Instruction-grain lifeguards are too slow. Unfortunately, existing approaches to instruction-grain lifeguards are very slow, because lifeguard functionality is invoked on (nearly) every instruction. Software-only solutions (mainly based on dynamic binary instrumentation, DBI [1, 14, 20]) typically slow down the monitored program by 10–100 times [17, 20, 33], limiting their utility to program debugging back at the development sites. To address this overhead, a number of hardware optimizations have been proposed, each tailored to a specific class of lifeguards: e.g., memory-access monitoring [28, 35, 42], data-race detection [41], and information-flow tracking with simple metadata [7, 8, 30, 34]. Unfortunately, these mechanisms are useful only for the narrow class of lifeguards that they support. Other studies [3, 5] have proposed more general-purpose hardware solutions. For example, our previous study proposed *Log-Based Architectures (LBA)* [3], which capture a log from a monitored program and ship it to another on-chip core that executes the monitoring functionality. An earlier study [5] proposed *Dynamic Instruction Stream Editing (DISE)*, which performs pattern-matching-based dynamic rewriting of a processor's instruction stream to insert calls to monitoring code. Both DISE and our earlier work on LBA focus only on reducing the costs of DBI, such as reducing the resource competition between monitored programs and lifeguards. As a result, the instruction-grain monitoring overhead, although significantly reduced, is still large (e.g., 3–5X slowdowns [3]).

Fast and flexible hardware acceleration. In this paper, we study hardware acceleration for speeding up a diverse range of instruction-grain lifeguards. Figure 1 depicts the general setting we consider, which reflects existing general-purpose lifeguard platforms such as DISE and LBA. On the left, an event-capture runtime observes the instructions executed by

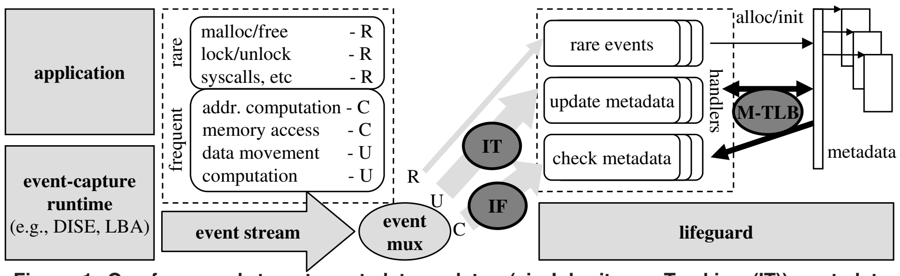


Figure 1. Our framework targets metadata updates (via Inheritance Tracking (IT)), metadata checks (via Idempotent Filters (IF)) and metadata mapping (via Metadata-TLB (M-TLB)).

	IT	IF	M-TLB
AddrCheck		yes	yes
MemCheck	yes	yes	yes
TaintCheck	yes		yes
TaintCheck w/ detailed tracking	yes		yes
LockSet		yes	yes

Figure 2. Applying our hardware acceleration framework to the lifeguards in this study. See Table 1 for lifeguard descriptions.

the monitored program, and creates a corresponding stream of event records. The dashed box on the left shows examples of rare and frequent events of interest. On the right, a lifeguard tracks the state of the application (e.g., which memory regions have been allocated) by maintaining application *metadata*. To consume the event stream, the lifeguard issues event handlers that may update its metadata, use the metadata to check the event against some invariant, or both.

This paper presents a hardware acceleration framework for instruction-grain lifeguards. By analyzing a number of lifeguards representing diverse monitoring requirements, we identify three main sources of overheads for instruction-grain lifeguards. Our framework provides novel techniques for addressing each of these sources:

(i) **Propagation tracking**. One key source of lifeguard overhead is *propagation tracking* (or *dynamic information flow tracking* [30]). Given an executed application instruction I, the metadata of I's destination location is computed as a combination of the metadata of all I's source locations (e.g., if one of I's sources contains "suspect" data, then I's destination now contains "suspect" data). Previous work on propagation tracking [7, 8, 21, 30] has considered only such *generic* propagation tracking, which is challenging to optimize because of the strong dependence among application instructions. Our key insights are that generic propagation tracking is more general than needed for important invariants, that a more restricted version suffices, and moreover, that such a version can be supported with significantly lower overhead using a novel technique called *Inheritance Tracking (IT)*.

(ii) **(Redundant) checking**. For many lifeguards, checking is performed on every memory reference and/or on every address computation. To reduce this second key source of overhead, our idea is to take advantage of the fact that lifeguards do not need to check "redundant" events, where the corresponding metadata are not changed. Hence, we design hardware *Idempotent Filters (IF)* to identify (and discard) redundant events without the overhead of accessing the metadata. Previously, redundant event filtering has been applied to address-checking lifeguards using other mechanisms [28], but we show how our technique can be applied more generally.

(iii) **Metadata mapping**. The third key source of overhead is the mapping from an application address to the corresponding metadata location(s) that is performed in almost every lifeguard event handler. We propose *Metadata-TLB (M-TLB)*, a novel mechanism that speeds up such metadata address translation. The M-TLB resides in user space. It can be configured at runtime for a variety of metadata sizes and organizations.

In this way, our framework achieves the performance of lifeguard-specific hardware approaches, while supporting a wide range of lifeguards, as illustrated in Figure 2.

Contributions. This paper makes the following main contributions. First, we analyze a number of diverse lifeguards to understand the requirements and commonalities of instruction-grain lifeguards, identifying three main sources of lifeguard overheads (Section 2). We use LBA as a case study lifeguard platform (Section 3). Second, we propose a hardware acceleration framework highlighted by three novel techniques for addressing these overheads: *Inheritance Tracking (IT)* for accelerating propagation-style metadata updates (Section 4), *Idempotent Filters (IF)* for identifying and discarding redundant checking events that will not alter metadata state (Section 5), and *Metadata-TLBs (M-TLB)* for accelerating the translation from application address to metadata address (Section 6). Finally, we implemented our techniques within LBA and evaluated them through simulation studies (Section 7). Our experiments with CPU-intensive benchmark programs (the most challenging case, because of their high instruction rate) show a 2–3X reduction in LBA's overheads for all of the studied lifeguards, down to a 2–51% overall slowdowns for all but one of the lifeguards.

2 Requirements for Efficiently Supporting Instruction-Grain Lifeguards

Our goal is to provide hardware accelerators for a wide range of instruction-grain lifeguards, addressing the major performance bottlenecks in such lifeguards while being sufficiently flexible to support their diverse requirements. This section analyzes a range of lifeguards in order to understand their commonalities and differences. We identify three significant, common sources of overheads for lifeguards, as well as five important axes on which lifeguards differ even in these common sources. Finally, we discuss related work in the context of these commonalities and differences.

2.1 Understanding Instruction-Grain Lifeguards

We focus on four diverse instruction-grain lifeguards that detect memory violations, security exploits, and data races, as detailed in Table 1. In the table, we describe for each lifeguard its purpose, the basic idea, the metadata maintained, and when/how metadata updates and checks are performed. To simplify the descriptions, we ignore checks that consume only a negligible fraction of the lifeguard's time.

From Figure 1 and Table 1, we see that execution events in the monitored program correspond to specific lifeguard

Table 1. Example instruction-grain lifeguards.

ADDRCHECK [17]	**Purpose:** Check whether every memory access is to an allocated region of memory. **Idea:** By intercepting memory allocation routines such as `malloc` and `free`, ADDRCHECK maintains metadata for each byte of the monitored program's address space indicating whether or not that byte is currently accessible. The metadata are checked on every memory operation for (mainly) invalid heap accesses and invalid stack accesses. **Metadata:** One "accessible" bit per address byte of the monitored program. **Metadata updates:** `malloc` and `free` events change the accessible state of the affected heap blocks. **Metadata checks:** For every memory access, check the accessible bit for the memory address. **Auxiliary structures:** A list of records for observed `malloc`'s and a list of records for observed `free`'s, which are used to detect double `free`'s, invalid `free`'s, and memory leaks.
MEMCHECK [18, 19]	**Purpose:** Extend ADDRCHECK to detect the use of uninitialized values. **Idea:** A memory load of an uninitialized value is not an error in itself (e.g., copying a partially initialized structure). Rather, an error is raised only if uninitialized values are dereferenced as pointers, used in conditional tests, or passed into system calls. To achieve this, MEMCHECK tracks the propagation of uninitialized values in the monitored program. **Metadata:** Same as ADDRCHECK + one "initialized" bit per address byte + "initialized" state per register. **Metadata updates:** Accessible bits are updated as in ADDRCHECK. Initialized bits are cleared after `free`'s and set for constant value writes and system call returns. Propagation tracking: For every executed instruction, the destination becomes uninitialized if at least one of the sources is uninitialized. **Metadata checks:** Accessible bits are checked as in ADDRCHECK. Initialized bits are checked for base/index registers of memory accesses, conditional test inputs, and system call inputs. **Auxiliary structures:** Same as ADDRCHECK.
TAINTCHECK [21]	**Purpose:** Detect overwrite-related security exploits (e.g., due to buffer overruns and format string vulnerabilities). **Idea:** All unverified program input data, such as data from the network, are marked as suspect, or *tainted*. Subsequently, the propagation of tainted data through the program is carefully tracked. If a tainted value is loaded from memory, the destination register is marked as tainted. A computation destination is marked as tainted if a source is tainted. An error is raised if tainted data are used in critical ways, such as in jump target addresses, format strings of `printf`-like calls, or system call arguments. **Metadata:** One "tainted" bit per address byte of the monitored program + "tainted" state per register. **Metadata updates:** Set "tainted" bits to untainted after `malloc`'s. Memory buffers used in `read` or `recv` system calls obtaining data from untrusted sources are marked as tainted. Tainted status is propagated for every executed instruction. **Metadata checks:** Check tainted status of register or memory locations of indirect jump target addresses, format strings of `printf`-like calls, and system call arguments. **TaintCheck with detailed tracking:** We also study a TAINTCHECK variant that records a history of the taint propagation, using an 8-byte metadata structure (4-byte "from" address, 4-byte instruction pointer) per 4-byte application word (see Section 7.1).
LOCKSET [27]	**Purpose:** Detect data races by checking whether accesses to shared memory locations are protected by consistent sets of locks. **Idea:** For each thread t, LOCKSET maintains the current set S_t of locks held by the thread. For each shared memory location m, it maintains a candidate set S_m of locks. m is known to be a shared location if a second thread accesses it; at this moment, S_m is initialized with the current lock set of the second thread. Afterwards, whenever a thread t references m, S_m is set to $S_m \cap S_t$. If S_m ever becomes empty, then no consistent common lock set protects accesses to m, and an error is raised. **Metadata:** A lockset is implemented as a sorted list of lock addresses. A 32-bit record is maintained for every 4-byte word in the monitored program. It consists of a compressed 30-bit pointer to the actual candidate lockset and a 2-bit state for the location (virgin, exclusive, shared read-only, shared read-write). If the state is exclusive, the 30-bit pointer is reused for recording its owner thread ID. For every thread, an uncompressed pointer to the thread's current lockset is maintained. **Metadata updates:** `malloc`'s set metadata to be virgin. A virgin location becomes exclusive after the first memory access. An exclusive location becomes shared after a memory access from a thread other than the owner. For shared locations, lockset intersection is performed for every memory access. **Metadata checks:** For every shared memory access, check if the resulting lockset after intersection is empty. **Auxiliary structures:** All the known locksets, each being a sorted list of lock addresses.

handler functionality, and that the interesting events are typically either important library/system calls or individual instructions. The former are often expensive to handle, but rare, while the latter are inexpensive individually, but so frequent that lifeguard performance is very sensitive to their cost.

Table 1 shows that the role of many instruction-grain event handlers centers around accessing lifeguard metadata (also called *shadow values* [20]), which represent the state information maintained by lifeguards regarding the monitored application's address space (e.g., which addresses have been tainted), often including application registers. Consequently, these handlers perform the following three major activities:

(1) **Metadata updates**: While some metadata (e.g., ADDRCHECK's accessible bits and LOCKSET's 32-bit records) are updated only at infrequent events such as library calls, others are updated at nearly every monitored instruction. The latter metadata arise in *propagation tracking* lifeguards such as MEMCHECK and TAINTCHECK, which propagate metadata status from sources to destinations on every instruction. Because each such update takes multiple instructions to perform, propagation tracking is a key source of lifeguard overhead.

(2) **Metadata checks**: Checks are often performed in instruction-grain events (e.g., for every application memory access in ADDRCHECK and MEMCHECK and every shared memory access in LOCKSET). Intuitively, in a well-behaved program, metadata converge into stable states quickly. This

insight can be exploited in lifeguard handlers by checking the frequent case—stable state—in a fast path while branching into a slow path for more detailed checks. However, even the most optimized checking operation has to do metadata access, comparison, and branch. Thus, (redundant) metadata checks are a second key source of lifeguard overhead.

(3) **Metadata mapping**: An address of the monitored application is mapped into a metadata location (e.g., an application address is mapped to an accessible bit in ADDRCHECK). This operation involves a sequence of mask and shift instructions, which often takes a significant portion of handler instructions (as high as half of the instructions, as will be illustrated in Section 6). Because this translation is required for every metadata check and update in all our lifeguards, metadata mapping is a third key source of lifeguard overhead.

As revealed by our experimental study in Section 7, these three sources of overheads constitute a significant fraction of the lifeguards' execution times.

Finally, as shown in Table 1, there are several important differences in the way that metadata are used by the four lifeguards. These differences fall along five axes: (i) unit for metadata (memory or register, per-byte or per-word), (ii) metadata bits per unit (1–64 bits); (iii) metadata semantics (e.g., LOCKSET's pointer plus state encoding); (iv) whether the metadata are propagation-tracking; and (v) use of auxiliary structures. These differences demand flexibility in the underlying support platform.

2.2 Related Work on Hardware Proposals for Instruction-Grain Lifeguards

Lifeguard-specific techniques. One class of prior hardware proposals achieves low overhead, but targets only one or a subset of lifeguards. HARD [41] implements hardware-based LOCKSET by extending every cache line with a hardware bloom filter representing the candidate lockset and augmenting the snoopy-based cache coherence protocol for lockset communication and updates. MemTracker [35] focuses only on memory access monitoring (propagation tracking, for instance, is not supported). The scheme adds an extra processor pipeline stage that performs lifeguard metadata updates and checks based on a state transition table with events such as load, store, alloc, and free. It cannot handle flexible metadata semantics such as that in LOCKSET. Similarly, several recent hardware proposals [7, 8, 30, 34] focus on improving the performance of propagation tracking—also known as *dynamic information flow tracking* (DIFT)—by introducing microarchitectural changes to enable the processor to manage metadata directly. While several of these proposals provide some policy flexibility (addressing axis (ii), above), they still restrict the metadata format and semantics (axes (i) and (iii)) to reduce the hardware complexity; hence, lifeguard generality is sacrificed. For example, they do not support non-DIFT lifeguards such as LOCKSET and cannot support propagation tracking with more flexible metadata such as TAINTCHECK with detailed tracking.

Techniques targeting binary instrumentation cost. The second class of hardware proposals strive to support a wide range of lifeguards. However, they focus only on reducing the costs incurred by dynamic binary instrumentation [1, 14, 20], the state-of-the-art software-only lifeguard platform. In particular, they seek to reduce (i) code translation costs for inserting event handlers into application code; (ii) competition for resources such as cycles, registers and caches; and/or (iii) the cost of re-creating hidden instruction states, such as memory addresses in IA32. DISE [5] removes cost (i) by extending the instruction fetch and decode unit with a macro-expansion capability for pattern-matching triggering events and inserting lifeguard event handlers on the fly. It also reduces cost (ii) somewhat by providing special registers for lifeguards. iWatcher [42] enables event handlers to be associated with address ranges, triggers event handlers based on cache line tags, and executes handlers and the monitored program in separate hardware threads, thus reducing all three costs. However, the scheme does not support propagation tracking or large tags. Several recent studies (Heapmon [28], INDRA [29], and LBA [3]) propose to take advantage of many-core processors and run lifeguards on otherwise idle cores. While Heapmon and INDRA studied only specific lifeguards, LBA is a general-purpose replacement for the dynamic binary instrumentation approach, reducing all three costs.

In contrast to this prior work, we provide hardware accelerators that address the three major performance bottlenecks in instruction-grain lifeguards, while being sufficiently flexible to support their diverse requirements.

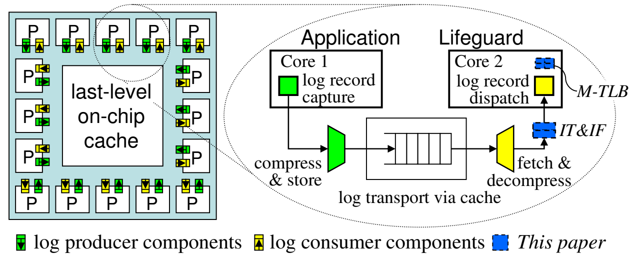


Figure 3. LBA on a many-core processor, including the components (IT, IF, M-TLB) proposed in this paper.

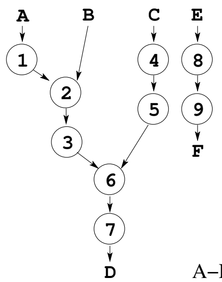
3 Case Study: Log-Based Architectures

As our work was done in the context of the Log-Based Architectures (LBA) project, we will use LBA as our running case study. Note, however, that our framework is not tied to LBA and can also be used to accelerate any lifeguard platform having the generic structure of Figure 1 (e.g., DISE [5]).

Figure 3 depicts a many-core processor enhanced with LBA producer components (darker/green rectangles with outgoing arrows) and LBA consumer components (lighter/yellow rectangles with incoming arrows) for every core. Given users' relative preferences of performance, power, and correctness, lifeguard monitoring can be dynamically enabled. In Figure 3, the dashed/blue rectangles on the consumer side are the new components proposed in this paper.

The zoom-in picture in Figure 3 shows that a lifeguard running on core 2 is monitoring an unmodified application running on core 1. As an application instruction retires, LBA captures a record, compresses it, and transports it through a buffer in on-chip cache. An instruction record (conceptually) consists of the program counter, instruction type, input/output

Application Data Flow



Example Application Instruction Sequence	Events Delivered To Lifeguard	TaintCheck Operation For Each Event	IT Filter State & Event after Filtering
(1) mov A, %eax	**mem_to_reg**	rtaint(%eax)=mtaint(A)	IT(%eax)=addr(A)
(2) add B, %eax	**dest_reg_op_mem**	rtaint(%eax) |= mtaint(B)	IT(%eax)=clear
(3) shr 8, %eax			
(4) mov C, %ecx	**mem_to_reg**	rtaint(%ecx)=mtaint(C)	IT(%ecx)=addr(C)
(5) and 0xff, %ecx			
(6) sub %ecx, %eax	**dest_reg_op_reg**	rtaint(%eax) |= rtaint(%ecx)	IT(%eax)=clear
(7) mov %eax, D	**reg_to_mem**	mtaint(D)=rtaint(%eax)	**imm_to_mem** (D)
(8) mov E, %eax	**mem_to_reg**	rtaint(%eax)=mtaint(E)	IT(%eax)=addr(E)
(9) mov %eax, F	**reg_to_mem**	mtaint(F)=rtaint(%eax)	**mem_to_mem** (E,F)

A–F: memory locations; rtaint/mtaint: taint values for register/memory; IT: filter state; **xx_to_xx**: delivered event

Figure 4. Propagation tracking for an example sequence of application instructions.

operand identifiers, and any data addresses (the compressed log records are less than a byte [3]). In addition, LBA supports software-inserted annotation records representing high-level events (e.g., `malloc` library calls), which can be captured via wrapper libraries. The consumer components support event-driven execution. A lifeguard is organized as a set of event handlers registered with LBA in an event type configuration table (ETCT); each instruction record corresponds to one or more events. Every handler ends with a special control transfer instruction—*nlba* (next LBA event). *nlba* does not have target addresses, instead it determines the event type from log records, looks up the ETCT, and changes program control to the entrance of the registered handler. Certain event values (such as data addresses) are automatically placed in registers for ready handler access.

LBA reduces the producer-consumer synchronization overhead by using a large (64KB–1MB) log buffer; however, if the buffer becomes full (empty), the application (lifeguard) must stall. Because of the decoupled nature of execution and checking, bug detection at the lifeguard lags bug occurrence at the application. LBA relies on OS level support for fault containment. The monitored application is stalled at each syscall until the lifeguard finishes checking all the remaining records in the log buffer; this prevents any damage from propagating into the OS kernel and affecting other applications.

4 Optimizing Propagation Tracking

Lifeguards such as TAINTCHECK and MEMCHECK (and generally all DIFT lifeguards [30]) track the propagation of data characteristics through the application's address space. While lifeguards that do not track data propagation may ignore certain classes of events (e.g., LOCKSET may ignore events that do not reference memory, such as register-register ALU operations), propagation-style lifeguards must track the flow of data through every operation that handles data, and hence must be triggered for nearly every event.

Consequently, propagation-tracking lifeguards often suffer high execution overhead. For example, Figure 4 shows a sequence of application instructions, the corresponding events, and the TAINTCHECK operations that would be triggered to handle them. Except for the two "self" operations, (3) and (5), an event is delivered for every application instruction. (See Figure 5 for the full list of propagation-tracking events.) Note that each TAINTCHECK operation may comprise multiple instructions. For example, the operation associated with event (1) must determine the metadata address associated with application address `A`, fetch `A`'s metadata (taint value), determine the metadata address associated with application register `%eax`, and finally store the taint value. Clearly, reducing the number of events that must be delivered to the lifeguard could result in significant performance savings.

4.1 Hardware-assisted Propagation Tracking

To improve the performance of propagation-tracking lifeguards, several studies [7, 8, 30, 34] have proposed hardware designs that automatically track metadata values. Unfortunately, these designs are quite narrow in scope—supporting only a single lifeguard or, at best, only a particular metadata size and organization (so that the hardware engine can access the lifeguard's metadata without software support). As a result, even simple modifications to lifeguards that perform well in their unmodified form, such as adding detailed tracking to TAINTCHECK, may reduce the lifeguard's performance from an acceptable level to a prohibitively low one.

As an alternative to propagating metadata *values* in hardware, we propose an optimization that tracks the data *inheritance* instead. The difference can be illustrated by returning to instruction (1) in Figure 4. A value propagation mechanism would require that hardware retrieve the metadata corresponding to memory address `A`, and associate it with register `%eax`. Inheritance tracking, in contrast, suggests that the hardware associate `%eax` with the *address* `A`, rather than its metadata. By separating the tracking of inheritance from the propagation of metadata values, we support a wider range of lifeguards because the hardware need not comprehend the metadata organization. Additionally, the lazy evaluation of the metadata often enables further optimizations, as we shall see later.

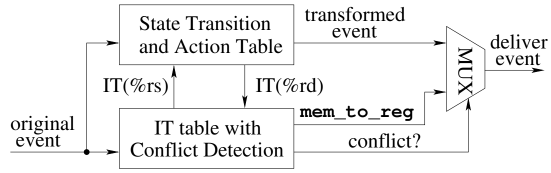
4.2 Unary Inheritance Tracking

Inheritance Tracking is particularly useful for eliminating events associated with the flow of data through registers. Consequently, an initial sketch for propagation performance acceleration would essentially be a small shadow register file that associates each architectural register with the addresses from which it inherits. The challenge with this design is that, with generic propagation, a particular register could have multiple ancestors. For example, after instruction (6) in Figure 4, `%eax`'s inheritance list contains `A`, `B`, and `C`.

Fortunately, we observe that in many situations, it is sufficient to track unary propagation instead of generic propagation. Here, unary propagation includes single-source/single-destination ("copy") operations (register-to-memory and memory-to-register), as well as binary computations that use an immediate value as a source operand. We assume that non-unary operations (those that combine more than one metadata source) propagate a "clean" result to the destination (e.g., *untainted* for TAINTCHECK or *initialized* for MEMCHECK). While at first thought this assumption may appear too liberal, we argue it is valid if (a) the lifeguard reports an error if a source of a non-unary operation is unclean, or (b) the semantics underlying the metadata values imply that, for all practical purposes, the result of a non-unary operation is a clean value. Perhaps surprisingly, both MEMCHECK and TAINTCHECK are candidates for unary inheritance tracking as MEMCHECK satisfies property (a) and TAINTCHECK satisfies property (b).

MemCheck. MEMCHECK monitors memory references to detect the use of unallocated locations or uninitialized values. To avoid false positives associated with the harmless copying of uninitialized memory values (e.g., in a padded `struct`), MEMCHECK does not report an error on the first load of an uninitialized value, but rather tracks the propagation of such values until one is used for a pointer dereference, conditional test input, or system call input [18, 19]. While this lazy evaluation of uninitialized value use is sufficient to eventually catch use of uninitialized memory, we argue that an eager evaluation that flags the first use of an uninitialized value in a non-unary computation is equally valid (e.g., flagging when uninitialized values are added). Our modified MEMCHECK checks the source operands of non-unary operations (identifying any uninitialized values as errors) and treats the destination operands as initialized, in order to avoid a cascading of error reports all based on the same uninitialized value.

TaintCheck. TAINTCHECK tracks the propagation of memory taint values to detect memory overwrite-based security exploits [21], and for all practical purposes, unary propagation appears to suffice for detecting such exploits. This claim is supported through four observations. First, the security literature [6, 37] reports that overwrite attacks (e.g., buffer overflow) rely almost exclusively on direct copying. Second, third-party analysts [31] often identify overwrite-based security vulnerabilities in proprietary software by causing a software crash through the introduction of a long input composed of a known pattern (e.g., repeating 0x55). A vulnerability is identified if the pattern is observed in expected locations in the core dump. This technique relies on a direct (unary) propagation of the input, and any identified vulnerability will be detected under unary-only propagation. Third, to empirically evaluate these claims, we analyzed the first six months of CVE security alert entries in 2007 [31]. For the entries involving open source software, we studied the source code patches and found that every memory overwrite vulnerability was due to unary propagation. Finally, while there is always a concern that attackers can specifically work around unary-only propagation, note that TAINTCHECK identifies attacks (including format string and function pointer attacks) *before* any attacking code executes. Thus, the attack is constrained to exploit the original application code, not any injected code. (This task could be made even more challenging by providing application developers with static analysis tools that identify when the structure of their code provides an opportunity for a non-unary attack.) For these reasons, we believe that assuming taintedness does not propagate through non-unary operations represents a good performance/coverage trade-off.



IT table

used for conflict detection

	IT state	addr	size	addr&~3	4bits	(addr&~3)+4	4bits
reg1	○ ○	○ ○	○ ○	○ ○ ○	○ ○	○ ○ ○	○ ○
regN	○ ○	○ ○	○ ○	○ ○ ○	○ ○	○ ○ ○	○ ○

IT state: 00 – clear, 01 – addr, 10 – in lifeguard

State Transition and Action Table

`Original Event` *example app. operation*	Starting IT(%rs) State: clean	addr	in lifeguard
`imm_to_reg` *mov $imm, %rd*	IT(%rd) := clean		
`imm_to_mem` *mov $imm, mem(daddr)*	**`imm_to_mem`**		
`reg_self` *op $imm, %rd*	do nothing		
`mem_self` *op $imm, mem(daddr)*	do nothing		
`reg_to_reg` *mov %rs, %rd*	IT(%rd) := IT(%rs)		IT(%rd):=IT(%rs) **`reg_to_reg`**
`reg_to_mem` *mov %rs, mem(daddr)*	**`imm_to_mem`**	**`mem_to_mem`**	**`reg_to_mem`**
`mem_to_reg` *mov mem(saddr), %rd*	IT(%rd) := saddr		
`mem_to_mem` *copy saddr to daddr*	**`mem_to_mem`**		
`dest_reg_op_reg` *op %rs, %rd*	do nothing	IT(%rd) := clean	
`dest_reg_op_mem` *op mem(saddr), %rd*	IT(%rd) := clean		
`dest_mem_op_reg` *op %rs, mem(daddr)*	do nothing	**`imm_to_mem`**	
`other` *other instructions*	flush all relevant registers' IT to "in lifeguard" by delivering **`imm_to_reg/mem_to_reg,`** then deliver **`other.`**		

Figure 5. A unary inheritance tracking design.

By limiting the inheritance tracking to unary propagation, each register in the inheritance table described above can be associated with at most one source—making such a structure feasible. Moreover, because non-unary operations act as "sinks" for propagation (the result is always clean), many propagation events can be eliminated. In Figure 4, for example, unary propagation removes the need for accessing the metadata for `A`, `B`, and `C`.

4.3 A Unary Inheritance Tracking Design

Figure 5 presents a hardware design for unary inheritance tracking. The events are chosen to support the IA32 architecture. Note, however, that one can remove several memory related events to support RISC architectures. We use a table, *IT*, to hold inheritance information for each general-purpose register. Each entry either specifies the memory address from which the register inherits or indicates that the register is "clean". (A third state, "in lifeguard", is discussed be-low.) For each incoming original propagation event, we look up the state transition and action table using the event type and the state of the source register. The action is either to update an entry in the IT table, to deliver an event to the lifeguard, or to simply discard the event. For example, if the event is a `reg_to_reg` type (*mov %rs, %rd*) and IT(%rs) holds an address *addr*, then IT(%rd) is updated to *addr*.

Three issues complicate the picture somewhat. First, because the inherited metadata values are lazily evaluated, a write-after-read conflict may occur if a "store to A" event (e.g., `imm_to_mem`) arrives when a register r currently inherits from A. A problem arises if the store event were delivered, updating A's metadata, and later a `reg_to_mem` event were to propagate r's metadata to another location B's metadata. B's metadata should inherit the previous value of A's metadata, which had been overwritten. We solve this potential problem by detecting the conflict and delivering a `mem_to_reg` event to the lifeguard just prior to the "store to A" event, so that the lifeguard can appropriately maintain A's previous metadata as the current state of r. $IT(r)$ is then set to "in-lifeguard", indicating that r's metadata is now maintained in software (until the next overwriting of register r).

Second, architectures such as IA32 support unaligned memory accesses and accesses with multiple sizes. To be conservative, we would like the conflict detection mechanism to match any recorded memory accesses that have overlapping ranges with the current store. As shown in Figure 5, the four rightmost columns in the IT table store a pair of 4-byte aligned addresses with bitmaps indicating used bytes. An incoming store can first match its 4-byte aligned addresses and then check the bitmaps to determine if there is a conflict.

Third, there may be instructions not represented by the explicit event types in Figure 5 (e.g., *xchg* in IA32). Since these instructions are typically less frequent, we deliver an `other` event where the lifeguard software can analyze the instruction record in a slow path to determine the appropriate action. Before delivering an `other` event, our unary tracking hardware flushes the IT state of relevant registers to ensure that the lifeguard has all the relevant up-to-date metadata states.

Moreover, we have optimized the state transition for binary operations with known "clean" *%rs*. In such cases, we choose to "do nothing", leaving the destination's metadata unmodified, which follows generic propagation.

In summary, because many of the events are processed solely by the IT mechanism and not by the lifeguard, the lifeguard overhead can be significantly reduced. For example, in Figure 4, IT reduces the number of delivered events from seven to two, as shown on the right of the figure.

5 Idempotent Filters for Checking Events

Checking metadata states upon observing certain application events is a fundamental operation of any lifeguard. While some lifeguards perform only a modest number of checks (e.g., TAINTCHECK), others perform checks very frequently (e.g., ADDRCHECK, MEMCHECK, and LOCKSET logically check every memory operation). However, many checks are idempotent (redundant). For example, once ADDRCHECK checks that a memory location is allocated, subsequent loads and stores to the same address need not be checked—until the next `free()` event. In this section, we present an Idempotent Filter design for reducing lifeguard checking overhead.

The idea is to introduce a lifeguard-configurable IF cache of recently observed checking events. If an incoming event hits in the cache, it is discarded (filtered). Upon a miss in the IF cache, an event E is delivered to the lifeguard. If E is configured to be cacheable, it is inserted into the IF cache with the LRU replacement policy.

Since different lifeguards have different checking requirements, the IF hardware extends the event type configuration table (ETCT), which specifies event handler addresses, to include the following fields that control the IF behavior. First, a cacheable bit specifies whether the lifeguard classifies the event as checking-only (non-updating). If set, events of that type can be filtered by the IF cache. Second, a "check categorization" (CC) field enables lifeguards to specify whether two event types result in the same checks (such as load and store events in ADDRCHECK). Third, there is a cacheable bit for every field of the instruction record. A line in the IF cache consists of the CC value and the set of selected record field values. The line is indexed by a hash code computed from the entire line. If the CC value and the selected fields of an incoming event match an existing cache entry, the IF hardware considers it a "hit" and assumes that the two events are idempotent. For example, ADDRCHECK would use the same CC value for load and store event types, and specify that the memory address and the size fields are cacheable. MEMCHECK employs IF similarly for accessibility checking. In contrast, LOCKSET must treat load and store operations as separate with respect to filtering by using different categorization values for them (surprisingly, IF does apply to LOCKSET despite only being able to observe a single thread at a time[1]).

Moreover, the ETCT specifies invalidation policies for the IF cache. Note that checks are only idempotent as long as the underlying metadata remain unmodified. If the relevant metadata changes, cached checks must be invalidated. We further augment the ETCT with two bits: one indicating whether an event of this type invalidates the entire IF cache (e.g., `malloc/free` calls or system calls), and one indicating whether the event invalidates records that match the specified CC value and selected fields of the event.

Perhaps most interestingly, we find that relatively small cache sizes (e.g., 32-entry) and associativity (e.g., 4-way) are remarkably useful for idempotent filtering. Extensive results are presented in Section 7.

6 Reducing the Cost of Metadata Mapping

As described in Section 2.1, instruction-grain lifeguards typically keep metadata for every byte or word in the address space of the monitored applications, and typically, this metadata is consulted and/or updated for each (unfiltered) memory reference event. Because metadata accesses are so frequent, providing a fast translation from application addresses

[1]For LOCKSET, we choose to invalidate the filter for all annotation records (including lock/unlock). Consider two redundant accesses to the same memory location m from the same thread t. Because there is no lock/unlock in between, t's lockset S_t at the two accesses must be the same. Intersecting m's candidate set S_m with the same S_t twice does not shrink S_m further.

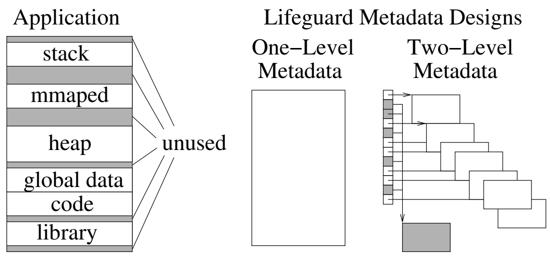


Figure 6. Two alternative lifeguard metadata designs. (We focus on the more flexible two-level design.)

to metadata addresses is essential for good lifeguard performance.

6.1 Lifeguard Metadata Organizations

From a survey of previous studies [7, 8, 17, 18, 21, 25, 27, 30, 35], we have identified the two possible metadata organizations shown in Figure 6. In the one-level design, a single contiguous metadata region represents a (possibly scaled) direct translation of the entire application address space. While this organization enables address translation through a simple scale-and-offset mechanism, it presents several drawbacks.

First, sparsely-used application address spaces consume an unnecessarily large portion of the lifeguard address space. Second, because both extremes of an application's virtual address space are typically occupied, the one-level design is only viable when metadata consume less space than normal data (e.g., 1 taint bit representing 1 byte), limiting its applicability. Third, the waste of metadata address space is particularly challenging when the application and lifeguard must share the same address space as in many proposals (including DISE); in face of the challenges of getting the application data, lifeguard metadata, lifeguard auxiliary data, and the lifeguard code to fit into a single virtual address space, robustness and portability considerations[2] often favor the two-level design [19].

The two-level design avoids these drawbacks by employing an indexing structure similar to a page table to perform the translation between application addresses and metadata addresses. This structure is clearly more space-efficient and flexible. The two-level design also requires fewer modifications to the normal loader, runtime, and OS behavior because the lifeguard can make better use of its available address space. Because our goal is to support a wide range of lifeguards, we consider the two-level design to be the baseline configuration.

6.2 Lifeguard-managed Address Translation

The one negative attribute of the two-level structure is performance, as the extra level of indirection requires additional lifeguard instructions and memory references. Figure 7 shows a representative event handler in TAINTCHECK, which combines the taint of a memory location and a register. The original C code is shown on the left along with the generated IA32 assembly instructions. Of the eight instructions, the first five performs metadata mapping, which is over half of the instructions in this handler! Our goal is to achieve the advantages of the two-level design while minimizing the cost.

[2] particularly given (1) large, uncooperative programs, or (2) tools which target multiple operating systems, and/or diverse hardware platforms.

Noting that the two-level metadata structure resembles page-tables, a hardware TLB mechanism suggests itself as a possible solution to the performance problem. Rather than translating virtual addresses to physical ones, however, such a mechanism would translate application-space virtual addresses to lifeguard-space (metadata) virtual addresses. Note that, because the output of the mechanism are lifeguard-space virtual addresses, the lifeguard can be permitted to manage the structure directly—minimizing (if not obviating) the need for OS support. In fact, a software-managed TLB [13] augmented to handle user-selectable metadata structure sizes would minimize the negative performance impact of the two-level design while preserving its flexibility.

6.3 The LMA (Load Metadata Address) Instruction

In the context of the IA32 architecture, we propose exposing a TLB-like address translation mechanism, which we call *Metadata-TLB (M-TLB)*, through a new instruction, *LMA* (Load Metadata Address), which translates an application address directly to a metadata address. Such a mechanism enables us to replace the first five instructions on the left side of Figure 7 with a single LMA instruction on the right side, thus reducing the handler's instruction count by half. Figure 8 describes the complete extension to the architecture.

Figure 9 depicts an M-TLB design. A lifeguard enables *LMA* support by configuring the metadata mapping with *lma_config*. Because the M-TLB is software managed (by the user-space lifeguard), a miss handler address is specified. An application data address is broken down into three parts: the highest part is the level1 index, the middle part is the level2 index, the lowest part is the index into each level2 element. The LMA config register records the number of level1 and level2 bits as well as the size of level2 elements. As shown in Figure 9, an *lma* instruction converts an application data address into a metadata address using a fast hardware lookup table (CAM). If successful, it takes one cycle. However, if the mapping is not found, the application address is pushed onto the stack and the software miss handler is called, which uses *lma_fill* to insert the missing mapping. When the miss handler returns, the *lma* instruction is re-executed.

We make the following three design choices. First, to support a wide range of lifeguards, our design is flexible in terms of the number of bits in level1 and level2 index and the element size. In fact, an *lma_config* flushes the M-TLB, and can be used to dynamically configure the mapping. Section 7.3 shows that such flexibility can significantly reduce M-TLB miss rates. Second, to reduce hardware complexity, *LMA* only performs address translation; it does not issue any memory accesses for metadata. Third, *LMA* only obtains the starting (byte) address of a level2 element; determining the offset of fields within an element (e.g., which bit corresponds to the taint of a given application byte address) is the responsibility of the lifeguard. In our experiences, this does not incur significant overhead because lifeguards can often use a level2 element as the most frequent metadata size, as in Figure 7.

7 Experimental Evaluations

We evaluate the performance benefits and design choices of our framework through both a simulation and a profiling

```
void dest_reg_op_mem_4B(UINT32 src_addr/*in %eax*/, UINT32 dest_reg/*in %edx*/)
  // App instruction event: dest_reg = dest_reg op mem(src_addr)
  // Handler performs:      reg_taint(dest_reg) |= mem_taint(src_addr)
```

```
map *mp = level1_index[src_addr>>16];
   // mov %eax, %ecx
   // shr $16,  %ecx
   // mov level1_index(,%ecx,4), %ecx
int idx = (src_addr & 0xffff) >> 2;
   // and $0xffff, %eax
   // shr $2, %eax
UChar mem_taint = mp[idx];
   // movzbl (%ecx,%eax,1), %eax
reg_taint[dest_reg] |= mem_taint;
   // or %al, reg_taint(%edx)
next_lba_record();
   // nlba
```

```
/* Applying LMA */

UChar *p = LMA_macro(src_addr);
   // LMA %eax, %ecx
UChar mem_taint = *p;
   // mov (%ecx), %al

reg_taint[dest_reg] |= mem_taint;
   // or %al, reg_taint(%edx)
next_lba_record();
   // nlba
```

Figure 7. Applying LMA to a TAINTCHECK event handler. (Two-level metadata structure: 16-bit level1 index, 14-bit level2 index, 2-bit in-byte offset. 2-bit tainted metadata is used so that the frequent 4-byte operations on IA32 are handled with 1-byte metadata accesses.)

lma_config $imm, $miss Set LMA config register and miss handler
lma %rs, %rt map an application address (%rs) into metadata address (%rt)
lma_fill %ra, %rb fill M-TLB with an entry given by application address (%ra) and metadata address (%rb)

Figure 8. New instructions for LMA.

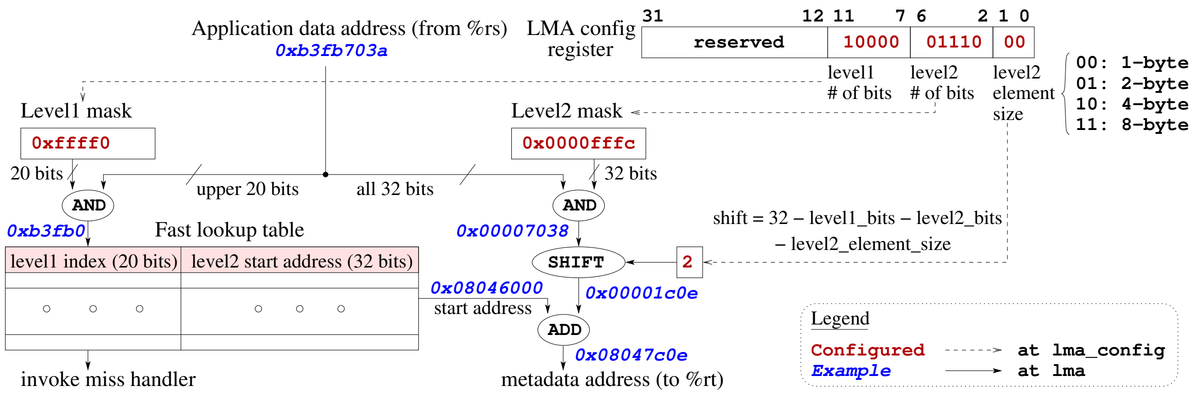


Figure 9. A 32-bit LMA hardware design illustrated with the TAINTCHECK example in Figure 7.

study. In the simulation study, we perform full-system simulations of five diverse instruction-grain lifeguards monitoring CPU-intensive benchmarks. However, due to simulation time constraints, reduced input sets are used. In contrast, we use the full-scale inputs in the profiling study, and investigate the impact of design choices on miss rates and filtered events.

7.1 Experimental Setup

Simulation Study. We implemented LBA by extending the Simics [36] full-system simulation platform with log record capture and event dispatch support. As described in Section 3, we extend our original LBA design with the capability of delivering multiple (hardwired) events per log record. The three proposed techniques, namely Inheritance Tracking (IT), Idempotent Filters (IF), and Load Metadata Address (LMA), are implemented in the event dispatch module, and are individually configurable by the lifeguard software.

Table 2 describes the simulation setup. We model a dual-core IA32 system with LBA, running an application on one core and a lifeguard monitoring the application on a second core. Because the choice of ISA may significantly affect the mix of monitored events, we choose the currently most popular ISA, IA32, in our study. Lacking a complete, publicly available model for out-of-order IA32 simulations, we model in-order cores. We simulate a two-level cache hierarchy with private L1 caches and a shared L2 cache, which contains a 64KB log buffer. The log buffer accesses are modeled as follows. To reduce L1 interference, a one-cache-line (64-byte) record buffer is used at both cores for caching compressed log records. The producer core writes an entire line of records to the L2 cache when the 64-byte buffer is filled. Likewise, the consumer core reads an entire line of records at a time.

In the simulation, we assume an 8-entry IT table (for 8 general-purpose registers), a 32-entry fully-associative cache for the IF filter, and a 1-cycle latency for the LMA instruction. We explore the design space in the profiling study.

Profiling Study. We instrument the benchmark executables with PIN [14] to obtain memory access, propagation, and address computation events. We build three modules that model the IT, IF, and LMA mechanisms, respectively. The modules take the events as input and collect statistics on miss rates and filtered events. We vary the design parameters of the techniques to explore the design space.

Lifeguard Implementations. We implemented five diverse lifeguards as shown in Figure 2. In addition to the common TAINTCHECK supported by previous lifeguard-specific hardware DIFT proposals [7, 8, 30, 34], we implemented a version of TAINTCHECK enhanced with detailed tracking. It keeps 8-byte metadata for every 4-byte application word. When a memory location is tainted by an instruction, the enhanced TAINTCHECK records the 4-byte "from" memory address and the 4-byte `eip`. In this way, a taint propagation trail can be reconstructed upon a security violation.

Table 2. Simulation Setup.

Simulator description		Simulation parameters	
Simulator	Virtutech Simics 3.0.17	Private L1I	16KB, 64B line, 2-way assoc, 1-cycle access lat.
Extensions	Log capture and dispatch	Private L1D	16KB, 64B line, 2-way assoc, 1-cycle access lat.
Target OS	Fedora Core 5 for IA32	Shared L2	512KB, 64B line, 8-way, 10-cycle access lat., 4 banks
Processor core	Dual-core, in-order scalar	Main Memory	200-cycle latency
Cache simulation	g-cache module	Log buffer	64KB, assuming 1B per compressed record [3]

Table 3. Multithreaded Benchmarks for LOCKSET.

Benchmark	Description and Input
blast v2.2.16 [16]	Searching a nucleotide and protein database of 134K sequences
pbzip2 v1.0.1 [22]	Parallel data compressor, compress half of CPU2000's ref input.source
pbunzip2 v1.0.1 [22]	Decompress pbzip2's output in parallel
water-nq SPLASH-2 [38]	Water simulation, 343 molecules
zchaff 2002.7.15 [23]	SAT (Boolean Satisfiability Problem) solver, circuit fault analysis

We strive to write the lifeguard code as optimized as possible. First, as shown previously in Figure 4, we use 2-bit metadata per application byte for the common TAINTCHECK. In this way, the frequent 4-byte operations in IA32 applications correspond to 1-byte metadata accesses, avoiding sub-byte access costs. MEMCHECK employs a similar scheme. Second, we optimize the code path for the frequent (stable) cases in metadata checking. For example, given a memory access, our LOCKSET implementation first checks if the 32-bit metadata is one of the recently seen stable cases[3] before performing the expensive lockset intersection operation. Third, we compile the lifeguards with gcc where event handlers are C functions with register parameters. We examine the disassembly of the frequent code path (typically less than 10 instructions) for the frequent event handlers. In cases that we see suboptimal code, we hand optimize the assembly code.

Benchmarks and Inputs. We choose CPU-intensive benchmarks to "stress test" instruction-grain monitoring. In the simulation study, we use the SPEC2000 integer benchmarks to evaluate all but the LOCKSET lifeguard. Because of the simulation time constraints, the *test* inputs are used. In contrast, we explore the design space in the profiling study with the *ref* inputs for the SPEC2000 integer benchmarks. For the data race detection lifeguard, LOCKSET, we use five multithreaded benchmarks in the simulation study, as shown in Table 3. Each benchmark spawns two working threads that are restricted to run on core 1 using Linux's `sched_setaffinity` call. Note that running multiple application threads on separate cores requires synchronizing both event delivery and lifeguard metadata access, and is beyond the scope of the paper. All benchmarks were run to completion in both the simulation and the profiling study.

7.2 Performance Study using Simulations

We evaluate the performance benefits of our framework using five diverse lifeguards. Figure 10 reports lifeguard performance with and without our proposed optimizations. The Y-axis reports the slowdowns of benchmark runs monitored by a lifeguard compared to benchmark runs without lifeguards. We see that compared to the LBA baseline bar, our techniques dramatically reduce the slowdowns for all the lifeguards, achieving less than 51% overhead on our challenging benchmarks for all but the MEMCHECK lifeguard.

The still poor performance of MEMCHECK is not surprising. Like ADDRCHECK, MEMCHECK checks memory accesses. Like TAINTCHECK, it propagates metadata. In addition, MEMCHECK also monitors other instruction-grain events, such as address computation for checking pointer dereference. Therefore, the events monitored by MEMCHECK is a super-set of ADDRCHECK and TAINTCHECK. Its slowdown is expected to be larger than the sum of ADDRCHECK's slowdown and TAINTCHECK's slowdown.

Figure 11 shows the performance improvement by applying one by one our three techniques (LMA for metadata mapping, IT for metadata updates, and IF for metadata checks). The Y-axis reports the average slowdowns across the benchmarks. As described in Figure 2, LMA is applicable to all the lifeguards, while IT and IF are relevant to a subset of the lifeguards. Moreover, IF is a flexible mechanism where a lifeguard can configure which event to filter and how to invalidate the filter. Both ADDRCHECK and LOCKSET require that memory accesses are filtered and the filter is invalidated at high-level events. However, they differ in that ADDRCHECK does not distinguish memory loads and stores while LOCKSET does. Like ADDRCHECK, MEMCHECK also checks memory accesses therefore employing the same filter. In addition, MEMCHECK also checks the base and index registers of address computations.

From Figure 11, we see that applying an additional technique achieves significant performance gains in all cases; the three techniques are complementary. This is because they are targeting different aspects of the lifeguard overhead. LMA reduces the average event handler lengths by replacing multiple metadata mapping instructions with a single LMA instruction. As shown in Figure 12, it reduces the total lifeguard dynamic instruction count by 16.7%–49.3%. IT and IF complement LMA by reducing the event frequency for update events and checking events, respectively. As shown in Figure 12, IT and IF filter out 24.9%–77.8% of the update and checking events.

7.3 Exploring Design Space using PIN Analysis

IT. We use PIN to instrument the SPEC2000 integer benchmarks with ref input. Figure 13(a) shows the percentage of reduced propagation events using Inheritance Tracking. IT removes 35.8%–82.0% of the propagation events.

IF. Figure 13(b) and (c) shows the average percentage of reduced check events by applying Idempotent Filters on the SPEC benchmarks with ref input while varying the number of filter entries and the set associativity of the filter design. Figure 13 (b) and (c) correspond to accessibility checking in ADDRCHECK and data race checking in LOCKSET, respec-

[3] That is, $S_m \cap S_t = S_m$, hence the metadata does not change.

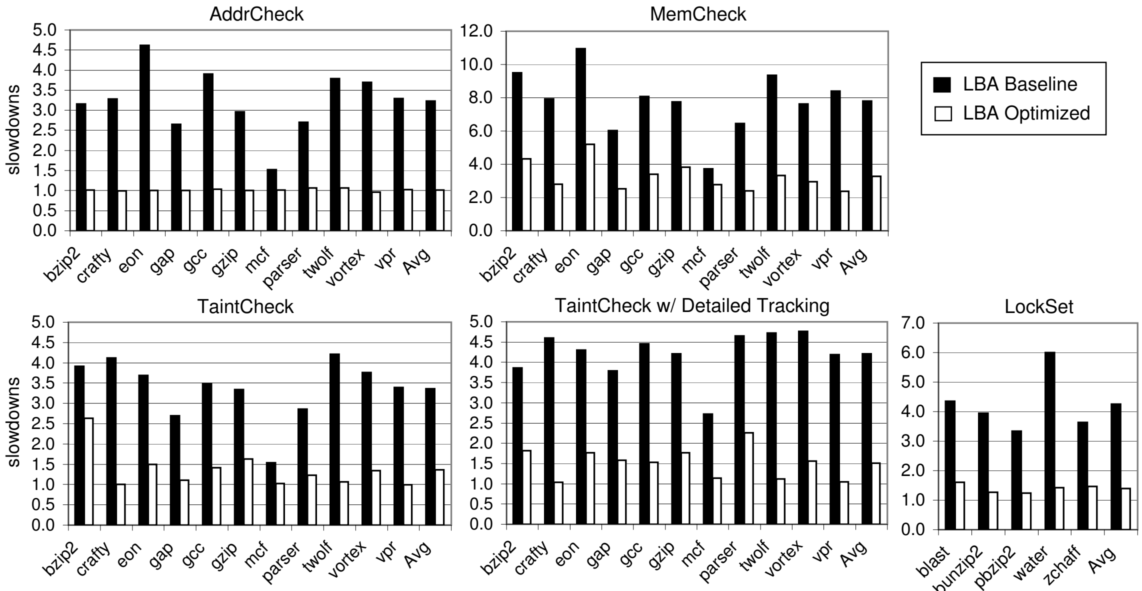


Figure 10. Performance benefits of our framework on five diverse lifeguards.

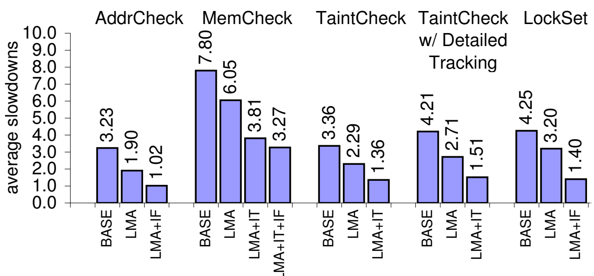
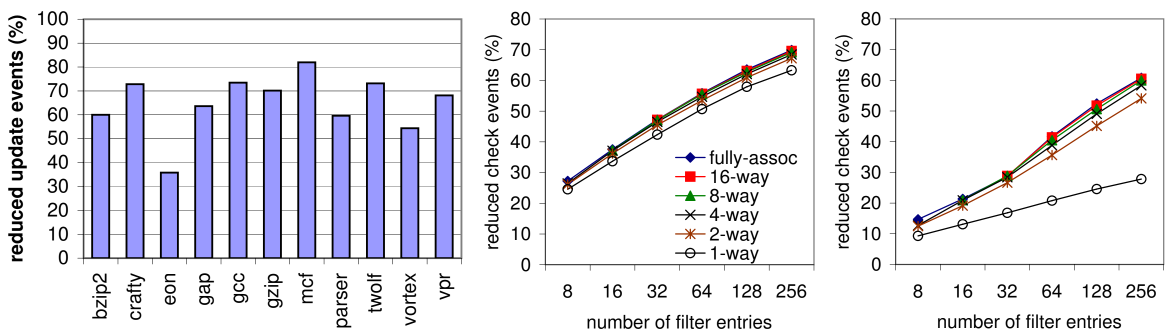


Figure 11. Applying our three techniques one by one.

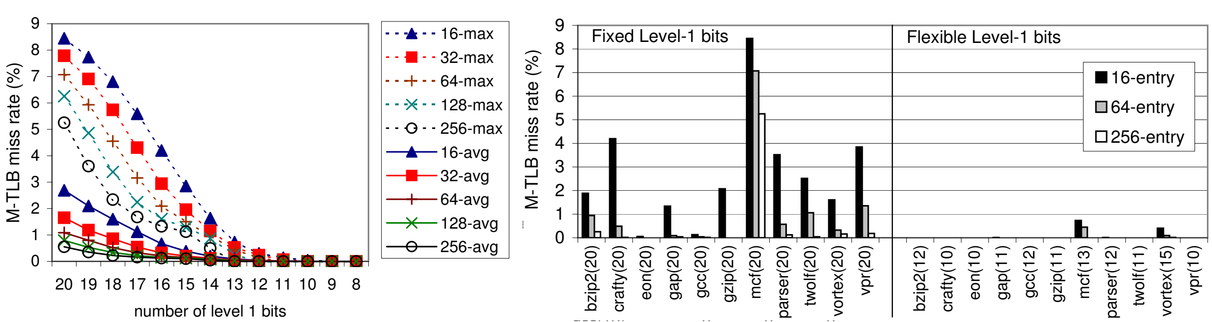
	LMA: Reduced dynamic instr	IT: Reduced update events	IF: Reduced check events
ADDRCHECK	28.9%–49.3%	–	38.2%–65.1%
MEMCHECK	16.7%–23.6%	24.9%–59.7%	38.2%–62.8%
TAINTCHECK	20.2%–42.4%	37.4%–74.4%	–
TAINTCHECK w/ detailed tracking	38.9%–44.3%	37.4%–74.4%	–
LOCKSET	18.2%–39.0%	–	43.5%–77.8%

Figure 12. Statistics on reduced instructions and events across the benchmarks.



(a) IT (b) IF for combined loads and stores (c) IF for separate loads and stores

Figure 13. IT and IF filtering for metadata updates and checks with PIN-based analysis.



(a) Varying number of level-1 bits (b) Fixed-bit vs. variable-bit design (level-1 bits in parenthesis)

Figure 14. Exploring the design space of M-TLB with PIN-based analysis.

tively. The difference is that the latter treats loads and stores separately while the former does not distinguish loads and stores. We see that IF can effectively reduce the number of check events, and a set-associative design with 4 or more ways works as well as the fully-associative design.

M-TLB. Figure 14(a) varies the number of level-1 bits from 20 to 8 bits in M-TLB and varies the number of M-TLB entries from 16 to 256. For every configuration, the maximum and the average miss rates of all benchmarks are depicted. As the number of level-1 bits decreases, the total number of level-1 entries decreases exponentially, thus resulting in the dramatic decrease of miss rates in the figure. On the other hand, lifeguard space overhead increases as level-1 bits decrease because small holes in the application space can no longer be distinguished. In Figure 14(b), we explore this trade-off and compare a fixed-level-1 design (left) with our flexible design (right). For the latter, the level-1 bits are chosen so that either the lifeguard space grows less than 10% or the lifeguard uses up to 1% of the total 32-bit address space (assuming a 1-1 mapping from application byte to metadata byte). We see that our flexible design dramatically reduces the M-TLB miss rates. In most cases, the miss rates are negligible.

8 Conclusion

This paper presented a hardware acceleration framework for a wide range of instruction-grain lifeguards. Three novel techniques—Inheritance Tracking, Idempotent Filters, and Metadata-TLBs—are shown to significantly reduce the major sources of overheads in such lifeguards. For all the lifeguards studied (but MEMCHECK), we found that these techniques reduced the overheads of LBA from 3.2–4.2X down to 1.02–1.5X on challenging CPU-intensive benchmarks. Our future work seeks to reduce these overheads further and also to study the performance on less challenging memory-bound benchmarks. The negligible overheads for our sole memory-bound benchmark (mcf) are encouraging in this regard.

Acknowledgments. This work is supported by grants from the National Science Foundation and from Intel. Ramachandran is supported in part by NSF grant CCF-0514876. We thank Anastassia Ailamaki, Limor Fix, Greg Ganger, Michelle Goodstein, Bin Lin, and Radu Teodorescu for their contributions and inputs to the LBA project.

References

[1] D. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, MIT, 2004.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice and Experience*, 30(7), 2000.

[3] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *ASID Workshop at ASPLOS*, 2006.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, 2001.

[5] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *ISCA*, 2003.

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

[7] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO*, 2004.

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

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2), 2001.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[12] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. of Computer Security*, 6(3), 1998.

[13] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[15] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[16] National Center for Biotechnology Information (NCBI). Basic local alignment search tool (BLAST). ftp://ftp.ncbi.nih.gov/blast/.

[17] N. Nethercote. *Dynamic binary analysis and instrumentation*. PhD thesis, U. Cambridge, 2004. http://valgrind.org.

[18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[19] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, 2007.

[20] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

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

[22] Parallel Bzip2. http://compression.ca/pbzip2/.

[23] Princeton Zchaff. http://www.princeton.edu/∼chaff/zchaff.html.

[24] N. Provos. Improving host security with system call policies. In *USENIX Security*, 2003.

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

[26] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies - a safe method to survive software failures. In *SOSP*, 2005.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM TOCS*, 15(4), 1997.

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

[29] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *ISCA*, 2006.

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

[31] The MITRE Corporation. Common vulnerabilities and exposures (CVE). http://cve.mitre.org/.

[32] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *SOSP*, 2007.

[33] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*, 2006.

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

[35] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA*, 2007.

[36] Virtutech Simics. http://www.virtutech.com/.

[37] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS*, 2003.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[39] M. Xu, R. Bodik, and M. D. Hill. A 'Flight Data Recorder' for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[40] M. Xu, R. Bodik, and M. D. Hill. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS*, 2006.

[41] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *HPCA*, 2007.

[42] Y. Zhou, P. Zhou, F. Qin, W. Liu, and J. Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM TACO*, 2(1), 2005.