# Flexible Hardware Acceleration for Instruction-Grain Lifeguards

Shimin Chen
Michael Kozuch
Phillip B. Gibbons
Michael Ryan
Intel Research Pittsburgh

Theodoros Strigkos
Todd C. Mowry
Olatunji Ruwase
Evangelos Vlachos
Carnegie Mellon University

Babak Falsafi
École Polytechnique Fédérale de Lausanne

Vijaya Ramachandran
University of Texas at Austin

INSTRUCTION-GRAIN LIFEGUARDS MONITOR EXECUTING PROGRAMS AT THE GRANULARITY OF INDIVIDUAL INSTRUCTIONS TO QUICKLY DETECT BUGS AND SECURITY ATTACKS, BUT THEIR FINE-GRAIN NATURE INCURS HIGH MONITORING OVERHEADS. THIS ARTICLE IDENTIFIES THREE COMMON SOURCES OF THESE OVERHEADS AND PROPOSES THREE TECHNIQUES THAT TOGETHER CONSTITUTE A GENERAL-PURPOSE HARDWARE ACCELERATION FRAMEWORK FOR LIFEGUARDS.

• • • • • • Systems designers have traditionally focused on maximizing performance, and more recently on minimizing power. From a user's perspective, however, both issues are of secondary concern when the software is misbehaving. 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. Although writing bug-free code has always been difficult, recent studies suggest that bug rates are getting worse over time as software complexity increases,[1] despite the software industry's prerelease 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.

There is a long history of developing tools to help diagnose and fix software problems at various phases of the software development and execution cycle: *static* tools attempt to identify problems before the program executes, *postmortem* tools attempt to reconstruct what went wrong after the application crashes, and *dynamic* tools—or *lifeguards*—monitor an application as it executes to diagnose and hopefully either contain or fix problems. *Instruction-grain lifeguards*, which perform invariant checking at the granularity of individual instructions, have two unique advantages. First, they have access to highly detailed information regarding dynamic events at the instruction level, such as memory references, address computations, and information flow. Second, software errors may be captured earlier and more accurately. The former enables a wide range of powerful lifeguards, for example, detecting memory-access violations, data races, and security exploits. The latter provides a better starting point for damage containment, on-site diagnosis,[2] and, hopefully, on-the-fly fixes and recovery.[3]

Unfortunately, software-only instruction-grain lifeguards, which are mainly based on dynamic binary instrumentation (DBI),[4,5] typically slow down the monitored program by 10 to 100 times,[4] because lifeguard functionality is invoked on nearly every instruction. To address this overhead, researchers have proposed several hardware optimizations, each tailored to a specific class of lifeguards: for example,

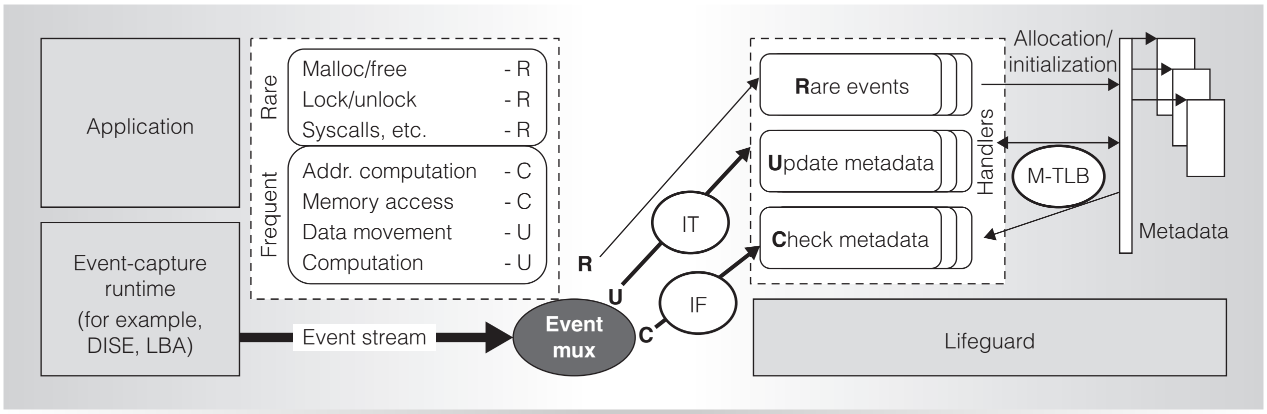


Figure 1. Our framework targets three common sources of lifeguard overheads: metadata updates via inheritance tracking (IT), metadata checks via idempotent filters (IF), and metadata mapping via metadata-TLB (M-TLB).

memory-access monitoring,[6-7] data-race detection,[8] and information-flow tracking with simple metadata.[9-11] However, each of these mechanisms is useful only for the narrow class of lifeguards that it supports. Other studies have proposed more general-purpose hardware solutions. For example, our previous study proposed *log-based architectures* (LBA)[12], which captures a log from a monitored program and ships it to another on-chip core that executes the monitoring functionality (see the "Log-Based Architectures" sidebar for more details on this approach). An earlier study[13] 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 (for example, 3 to 5× slowdowns[12]).

We believe that the key to long-term impact is fast and flexible lifeguards, for several reasons. First, lifeguards that are too slow will not be used in the field, as we observe today. Second, the set of important applications continues to change significantly over time. Third, researchers are actively creating increasingly sophisticated lifeguard algorithms to model and track increasingly subtle and complex software correctness problems. Finally, and perhaps most importantly, security attacks evolve quickly over time in response to new defenses; hence, we need software's flexibility to adapt to these quickly changing threats. Therefore, our goal in this article is to provide hardware accelerators for a wide range of instruction-grain lifeguards.

We analyze several diverse lifeguards (see the "Example Instruction-Grain Lifeguards" sidebar for their descriptions) and identify three main sources of lifeguard overheads. Then, we propose a hardware acceleration framework highlighted by three novel techniques for addressing these overheads: *inheritance tracking* (IT) for accelerating propagation style metadata updates, *idempotent filters* (IF) for accelerating metadata checks, and *metadata-TLBs* (M-TLB) for accelerating the translation from application address to metadata address. Finally, we implement our techniques within LBA and evaluate them through simulation studies. Experimental results show that our framework reduces overheads by 2 to 3× for all of the studied lifeguards over the previous state-of-the-art—down to 2 to 51 percent overall slowdowns for all but one of the lifeguards.

## Analyzing instruction-grain lifeguards

Figure 1 depicts the general setting we consider, reflecting existing general-purpose lifeguard platforms such as DISE and LBA. On the left, an event-capture runtime observes the instructions executed by the monitored program and creates a corresponding stream of event records.

## Log-Based Architectures: Exploiting Multicores for Lifeguards

Although multicore scaling makes software reliability more challenging (because of the challenges of parallel programming), it also provides the much-needed hardware resources to include architectural support for instruction-grain monitoring. Log-based architectures (LBA) exploits multicore resources to offer a general-purpose lifeguard support platform that runs the monitored application and the lifeguard on separate cores. Given users' relative preferences of performance, power, and correctness, LBA lets users dynamically enable lifeguard monitoring.

Figure A shows the LBA components, with a lifeguard running on core 2 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 the on-chip cache. An instruction record consists of the program counter, instruction type, input/output operand identifiers, and any data addresses (the compressed log records are less than a byte). In addition, LBA supports software-inserted annotation records representing high-level events (such as `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). Every handler ends with a special instruction—*nlba* (next LBA event)—which examines log records, looks up the ETCT, and transfers program control to the registered handler for the next event. 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 log buffer (64 Kbytes to 1 Mbyte); however, if the buffer becomes full, the application must stall, and if the buffer becomes empty, the lifeguard must stall. Because of the decoupled 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 syscalls until the lifeguard finishes checking the remaining records in the log buffer; this prevents any damage from propagating into the OS kernel and affecting other applications.

Compared to dynamic binary instrumentation, which runs a version of the application code that is dynamically modified to embed lifeguard functionality, LBA significantly reduces the performance overhead by avoiding the contention between the application and lifeguard functions for per-core hardware resources (such as registers and the L1 cache), and by streamlining event capture and delivery. Compared to hardware mechanisms specialized for particular types of lifeguards, LBA targets lifeguards written in software, and thus achieves the flexibility for supporting a wide range of existing and emerging lifeguards.

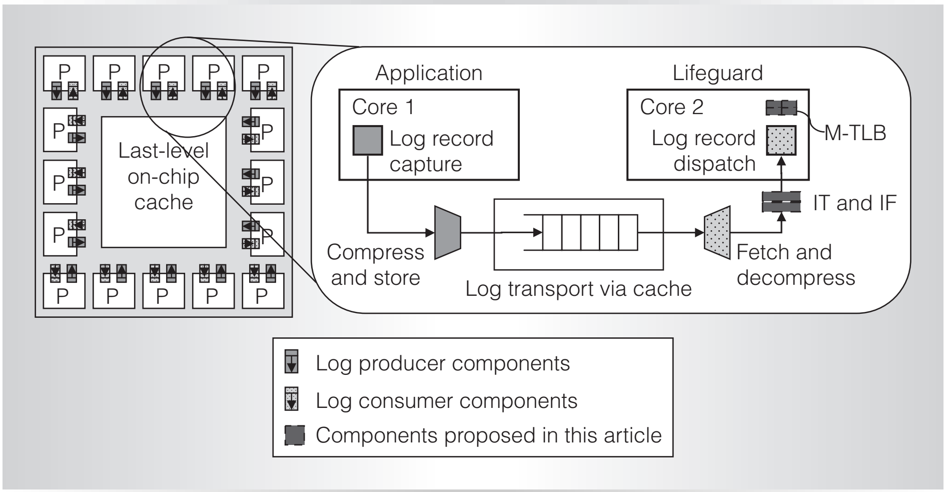


Figure A. Log-based architectures (LBA) on a multicore processor, including the components proposed in this article.

## Example Instruction-Grain Lifeguards

We focus on the following five diverse instruction-grain lifeguards that detect memory violations, security exploits, and data races in our study.

### AddrCheck

AddrCheck checks whether every memory access is to an allocated region of memory.[1] By intercepting memory allocation routines such as `malloc` and `free`, AddrCheck maintains one-bit metadata for each byte of the monitored program's address space indicating whether or not that byte is currently accessible. AddrCheck checks the metadata for every memory access.

### MemCheck

MemCheck extends AddrCheck to detect the use of uninitialized values.[2] For this purpose, it maintains one ''initialized'' bit per address byte and the ''initialized'' state per register in addition to the ''accessible'' bit. A memory load of an uninitialized value is not an error in itself (for example, copying a partially initialized structure). Rather, MemCheck raises an error 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: For every executed instruction, the destination becomes uninitialized if at least one of the sources is uninitialized.

### TaintCheck

TaintCheck detects overwrite-related security exploits, such as buffer overflow and format string attacks.[3] The metadata is one ''tainted'' bit per address byte of the monitored program and the ''tainted'' state per register. TaintCheck marks all unverified program input data, such as data from the network, as suspect, or *tainted*. Subsequently, it carefully tracks the propagation of tainted data through the program. If a tainted value is loaded from memory, TaintCheck marks the destination register as tainted. It marks a computation destination as tainted if a source is tainted. It raises an error if the program uses tainted data in critical ways, such as in jump target addresses, `printf`-like format strings, or 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. Upon detection, this lifeguard can reconstruct a taint propagation trail.

### LockSet

LockSet detects data races by checking whether the monitored program follows a consistent locking principle.[4] 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. LockSet knows $m$ 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, no consistent common lock set protects accesses to $m$, and LockSet raises an error. A LockSet structure is a list of lock addresses. For every 4-byte word in the monitored program, the metadata is a 32-bit record consisting of a compressed 30-bit pointer to the actual LockSet and a 2-bit state (indicating virgin, exclusive, shared read-only, or shared read-write) for the location.

### References

1. N. Nethercote, *Dynamic Binary Analysis and Instrumentation,* doctoral dissertation, Trinity College, Univ. of Cambridge, 2004.
2. N. Nethercote and J. Seward, ''How to Shadow Every Byte of Memory Used by a Program,'' *Proc. Int'l Conf. Virtual Execution Environments* (VEE 07), ACM Press, 2007, pp. 65-74.
3. J. Newsome and D. Song, ''Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software,'' *Proc. Network and Distributed System Security Symp.* (NDSS 05), The Internet Society, 2005; http://www.isoc.org/isoc/conferences/ndss.
4. S. Savage et al., ''Eraser: A Dynamic Race Detector for Multi-Threaded Programs,'' *ACM Trans. Computer Systems,* vol. 15, no. 4, Nov. 1997, pp. 391-411.

The dashed box on the left shows examples of rare and frequent events of interest. On the right, a lifeguard tracks the application's state (for example, which memory regions have been allocated) by maintaining *metadata* regarding the monitored application's address space, often including application registers. To consume the event stream, the lifeguard issues event handlers that can update its metadata, use the metadata to check the event against some invariant, or do both.

As Figure 1 and the lifeguard descriptions in the sidebar show, the role of many instruction-grain event handlers centers on accessing lifeguard metadata. We have identified three main sources of overheads for instruction-grain lifeguards as follows.

### Propagation-style metadata updates

Whereas some metadata (such as AddrCheck's accessible bits) are modified only at infrequent events such as library calls, others are updated at nearly every monitored instruction. These more frequently modified types of metadata arise in *propagation-tracking* (also known as dynamic information flow tracking[9-11]) lifeguards such as MemCheck and TaintCheck, which propagate metadata states from sources to destinations on every instruction. Because each metadata update takes multiple instructions to perform, propagation tracking is a key source of lifeguard overhead.

### Metadata checks

Lifeguard event handlers often perform checks for instruction-grain events (for example, for every application memory access in AddrCheck and MemCheck, and for every shared memory access in LockSet). Intuitively, in a well-behaved program, metadata converge into stable states quickly. Lifeguard event handlers can exploit this insight 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 must perform metadata access, comparison, and branch. Thus, metadata checks are a second key source of lifeguard overhead.

### Metadata mapping

Metadata mapping occurs when a lifeguard event handler maps an address of the monitored application into a metadata location (for example, when AddrCheck maps an application address to an accessible bit). 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 in the example lifeguards). 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.

### Differences in metadata use demand flexibility

Finally, as the lifeguard descriptions make clear, there are several important differences in the way that lifeguards use metadata:

- metadata unit (memory or register, per-byte or per-word);
- metadata bits per unit (1 to 64 bits);
- metadata semantics; and
- whether metadata require propagation tracking.

These differences demand flexibility in the underlying support platform.

## Inheritance tracking for propagation-style metadata updates

Propagation-style lifeguards must be triggered for nearly every instruction event for tracking data flow, yielding high monitoring overhead. Previous studies proposed hardware designs that extend processor pipelines to automatically access and propagate metadata along with the operations in the monitored program.[9-11] A fundamental limitation in these approaches is that because the hardware must understand metadata formats, simplifying assumptions about metadata must be made, so that the hardware supports only a single lifeguard or, at best, only a particular metadata size and organization. As a result, even simple modifications to lifeguards that perform well in their unmodified form, such as the addition of detailed tracking to TaintCheck, can reduce the lifeguard's performance from an acceptable level to a prohibitively low one.

We address this limitation by exploiting a common feature of propagation-style lifeguards: Metadata propagations follow the monitored program's dataflow structure.

We propose tracking data *inheritance* instead of propagating metadata *values* in hardware. For example, consider a monitored instruction "`mov A, %eax`", which moves the data in memory location `A` into register `%eax`. A value-tracking mechanism would move `A`'s metadata to `%eax`'s metadata, whereas inheritance-tracking hardware records that `%eax` inherits from `A`. By separating the tracking of inheritance from the propagation of metadata values, the hardware does not need to comprehend the metadata organization, thereby supporting a wider range of lifeguards.

### Unary inheritance tracking

Inheritance tracking is particularly useful for eliminating events associated with the flow of data through registers. Consequently, an initial sketch employs a small shadow register file that associates each architectural register with the addresses from which it inherits. However, with generic propagation, a particular register can inherit from multiple ancestors, and the number of ancestors can grow exponentially.

Fortunately, in many situations, we can track unary propagation instead of generic propagation. Here, unary propagation includes single-source, single-destination ("copy") operations, as well as binary computations that use an immediate value as a source operand. We assume that nonunary operations (those that combine more than one metadata source) propagate a "clean" result to the destination. Although at first thought this assumption might appear too liberal, we argue it is valid if

1. the lifeguard reports an error if a source of a nonunary operation is unclean, or
2. the semantics underlying the metadata values imply that, for all practical purposes, the result of a nonunary operation is a clean value.

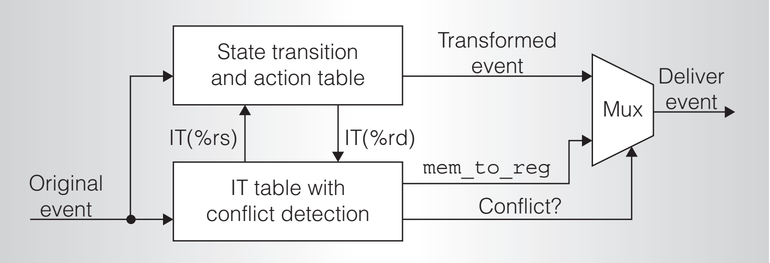
Perhaps surprisingly, both MemCheck and TaintCheck are candidates for unary inheritance tracking: MemCheck satisfies property 1, and TaintCheck satisfies property 2.

### Unary IT for MemCheck

MemCheck tracks the propagation of uninitialized values for avoiding false positives and reports errors when an uninitialized value is used for a pointer dereference, conditional test input, or system call input. Although this lazy evaluation eventually catches the use of uninitialized memory, an eager evaluation that flags the first use of an uninitialized value in a nonunary computation is equally valid (for example, flagging when uninitialized values are added). MemCheck modified in this way checks the source operands of nonunary operations and treats the destination operands as initialized, thus avoiding a cascade of error reports all based on the same uninitialized value.

### Unary IT for TaintCheck

TaintCheck tracks the propagation of memory taint values to detect memory overwrite-based security exploits. For all practical purposes, unary propagation provides good support for detecting such exploits. First, the literature on security reports that overwrite attacks (such as buffer overflow) rely almost exclusively on direct copying.[14] Second, third-party analysts typically 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 (such as repeating 0×55). A vulnerability is identified if the pattern is observed in specific locations in the core dump. Because this technique relies on a direct (unary) propagation of the input, any identified vulnerability (present and in the future) will be protected by unary-only propagation. Third, we analyzed the first six months of MITRE's Common Vulnerabilities and Exposures security alert entries in 2007 (CVE, http://cve.mitre.org). 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, although there is always a concern that attackers can specifically work around unary-only propagation, TaintCheck identifies attacks before any attacking code executes. Thus, the attack is constrained to exploit the original application code, not any injected code, which is substantially more challenging to proceed against. For these reasons, we believe that assuming taintedness does not propagate through nonunary



IT table — Used for conflict detection

| | IT state | addr | size | addr&~3 | 4bits | (addr&~3)+4 | 4bits |
|---|---|---|---|---|---|---|---|
| reg1 | o o | o o | o o | o o o | o o | o o o | o o |
| regN | o o | o o | o o | o o o | o o | o o o | o o |

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

Figure 2. A unary inheritance-tracking design.

operations represents a good performance/coverage trade-off.

### A unary IT design

By limiting inheritance tracking to unary propagation, each register in the inheritance table described earlier can be associated with at most one source—making such a structure feasible. Figure 2 presents a hardware design for unary inheritance tracking. We use the IT table 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,'' means that the hardware has delivered the register state to the lifeguard. 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. The conflict detection logic detects any incoming event `E` whose destination address overlaps existing inherit-from addresses. In such cases, the affected IT states are delivered to the lifeguards (with "`mem_to_reg`" events) before event `E`, thus guaranteeing that lifeguards always see the correct event order.

Because the IT mechanism processes many of the events without invoking the lifeguard, this design can significantly reduce the lifeguard overhead.

## Idempotent filters for metadata checks

Checking metadata states upon observing certain application events is a fundamental operation of any lifeguard. Whereas some lifeguards, such as TaintCheck, perform only a modest number of checks, others perform checks frequently—AddrCheck, MemCheck, and LockSet check every memory operation. However, many checks are idempotent (and thus 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.

Thus, we designed an idempotent filter (IF) to reduce 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.

Because different lifeguards have different checking requirements, the IF hardware extends the event type configuration table (ETCT), which specifies event handler addresses, to include several fields that control IF behavior. First, a cacheable bit specifies whether the lifeguard classifies the event as checking-only (nonupdating). If set, events of that type can be filtered by the IF cache. Second, a check categorization (CC) field lets lifeguards 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 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 separately with different

```
void dest_reg_op_mem_4B (UINT32 src_addr, UINT32 dest_reg)
 // app event type: dest_reg = dest_reg  op  mem(src_addr)
 // handler: reg_taint(dest_reg)|=mem_taint(src_addr)
 // src_addr is in %eax, dest_reg is in %edx
```

```
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 3. Applying the load metadata address (LMA) instruction to a TaintCheck event handler. The two-level metadata structure consists of a 16-bit level-1 index, a 14-bit level-2 index, and a 2-bit in-byte offset. TaintCheck uses 2-bit tainted metadata so that the frequent 4-byte operations on IA32 are handled with 1-byte metadata accesses.

categorization values. (Surprisingly, IF does apply to LockSet because intersecting with a set twice does not shrink the result further.)

Moreover, the ETCT specifies invalidation policies for the IF cache. 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 (for example, `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 (for example, 32 entries) and set-associativity (for example, 4-way) are remarkably useful for idempotent filtering in our experiments.

## Metadata-TLB for metadata mapping

Instruction-grain lifeguards keep metadata for every byte or word in the address space of the monitored applications that is consulted, updated, or both for each (unfiltered) memory reference event. For space efficiency and flexibility, such lifeguards often use a two-level metadata organization,[4] which employs an indexing structure similar to a page table to perform the translation between application and metadata addresses.

However, the one negative attribute of the two-level structure is performance, because the extra level of indirection requires additional lifeguard instructions and memory references. Figure 3 shows a representative event handler in TaintCheck that combines the taint of a memory location and a register. The left side of the figure shows the original C code along with the generated IA32 instructions. Of the eight instructions, the first five perform metadata mapping, accounting for more than half of the instructions in this handler! Our goal is to achieve the advantages of the two-level design while minimizing the cost.

Noting that the two-level metadata structure resembles page tables, we propose a hardware translation look-aside buffer mechanism, *metadata-TLB* (M-TLB), for solving the performance problem. Rather than translating virtual addresses to physical ones, M-TLB translates application-space virtual addresses to lifeguard-space (metadata) virtual addresses. An application data address consists of three parts: the highest part is the level-1 index, the middle part is the level-2 index, and the lowest part is the index into each level-2 element. The level-1 index is used to look up M-TLB for the

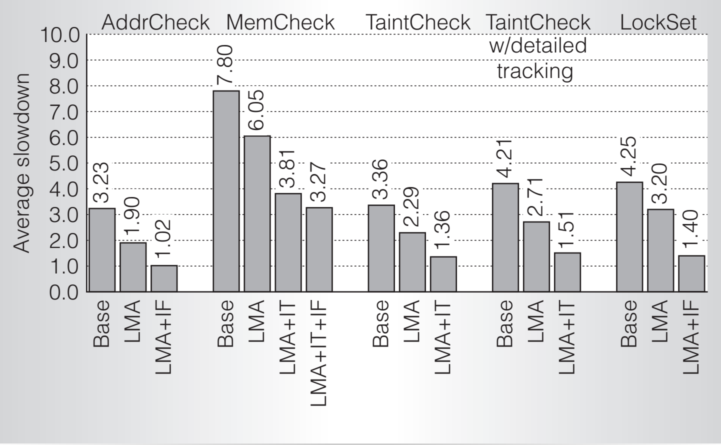


Figure 4. Performance benefits of applying inheritance tracking (IT), idempotent filters (IF), and load metadata address (LMA) to five diverse lifeguards.

starting address of the level-2 metadata chunk. Simple hardware structures perform the required shift and mask operations. Moreover, M-TLB is managed by lifeguards, minimizing the need for OS support: Upon an M-TLB miss, a configured software handler is called to insert new entries into the M-TLB.

In the context of the IA32 architecture, we propose to expose M-TLB through a new instruction, `lma` (load metadata address), which translates an application address and puts the metadata address into a specified register. As Figure 3 shows, such a mechanism lets us replace the first five instructions (left side) with a single `lma` instruction (right side), thus reducing the handler's instruction count by half.

We made the following design choices. First, to reduce hardware complexity, `lma` only performs address translation; it does not issue any memory accesses for metadata. Second, to support flexible lifeguard metadata, our design lets lifeguards configure the mapping parameters with a `lma_config` instruction. Third, for the same reason, `lma` only computes the starting (byte) address of a level-2 element; the lifeguard must determine the offset of fields within an element (for example, which bit corresponds to the taint of an application byte). In our experience, this does not incur significant overhead, because, as Figure 3 shows, lifeguards can often use an entire level-2 element as the most frequently accessed size.

## Experimental evaluations

We implemented LBA by extending the Virtutech Simics full-system simulation platform with log record capture and event dispatch support. The three proposed techniques—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.

Our simulation models a dual-core IA32 system with a two-level cache hierarchy augmented with LBA, running an application on one core and a lifeguard monitoring the application on a second core. The log buffer occupies 1/8 of the L2 cache. In the simulation, we assumed an 8-entry IT table (for eight general-purpose IA32 registers), a 32-entry fully-associative cache for the IF filter, and 1-cycle latency for the LMA instruction. The simulation also models interference of log accesses to the application and lifeguard.

We implemented five diverse lifeguards: AddrCheck, MemCheck, TaintCheck, TaintCheck with detailed tracking, and LockSet. We chose CPU-intensive SPEC2000 integer benchmarks to "stress test" instruction-grain monitoring. We used the *test* inputs due to simulation time constraints. For the data race detection lifeguard, LockSet, we chose five CPU-intensive multithreaded benchmarks and ran all the application threads on core 1. We ran all benchmarks to completion.

Figure 4 shows the performance improvements we achieved by applying our three techniques one by one (LMA on top of M-TLB for metadata mapping, IT for metadata updates, and IF for metadata checks). Compared with the LBA baseline, our framework achieves a 2 to 3× reduction in LBA's overheads for all of the studied lifeguards, with overall slowdowns of 2 to 51 percent for all but MemCheck. Because MemCheck performs propagation, checks memory accesses, and checks binary operands, its cost is larger than the sum of AddrCheck and TaintCheck, as expected.

Moreover, we evaluate the design parameters of our techniques with a profiling study using Pin.[5] We instrumented the benchmark executables to obtain memory access, propagation, and address computation events.

We built three Pin modules that model the IT, IF, and M-TLB mechanisms. The modules take the events as input and collect statistics on miss rates and filtered events. In contrast to the simulation study, we used the full *ref* input for SPEC2000 integer benchmarks. Our profiling results show that IT removes 35.8 percent to 82.0 percent of the propagation events, that an IF design with a set associativity of 4 or more works as well as the fully-associative design, and that our M-TLB design with flexible level-1 bits achieves less than 1 percent M-TLB miss rates.

Software complexity is increasing to unprecedented levels, commensurate with hardware design size and complexity, and system robustness is becoming critical as computers tackle day-to-day problems with increasing demands on reliability and security. Instruction-grain lifeguards offer a promising solution to this confluence of challenges. Our hardware acceleration framework—employing inheritance tracking, idempotent filters, and metadata TLBs—can significantly reduce the major sources of overheads in such lifeguards. Combined with LBA, our framework is the first major work to show a fully general-purpose lifeguard framework with low enough overheads to be deployed in the field without resorting to sampling. Our work presents a firm step toward addressing the long-term challenge of software reliability. MICRO

## Acknowledgments

Grants from the US National Science Foundation and from Intel supported this work. 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 to the LBA project.

## References

1. A. Chou et al., ''An Empirical Study of Operating Systems Errors,'' *Proc. ACM Symp. Operating Systems Principles* (SOSP 01), ACM Press, 2001, pp. 73-88.
2. J. Tucek et al., ''Triage: Diagnosing Production Run Failures at the User's Site,'' *Proc. ACM Symp. Operating Systems Principles* (SOSP 07), ACM Press, 2007, pp. 131-144.
3. F. Qin et al., ''Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures,'' *ACM Symp. Operating Systems Principles* (SOSP 05), ACM Press, 2005, pp. 235-248.
4. N. Nethercote and J. Seward, ''Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,'' *Proc. Conf. Programming Language Design and Implementation* (PLDI 07), ACM Press, 2007, pp. 89-100.
5. C.-K. Luk et al., ''Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,'' *Proc. Conf. Programming Language Design and Implementation* (PLDI 05), ACM Press, 2005, pp. 190-200.
6. G. Venkataramani et al., ''MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging, *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 07), IEEE CS Press, 2007, pp. 273-284.
7. Y. Zhou et al., ''Efficient and Flexible Architectural Support for Dynamic Monitoring,'' *ACM Trans. Architecture and Code Optimization,* vol. 2, no. 1, Mar. 2005, pp. 3-33.
8. P. Zhou, R. Teodorescu, and Y. Zhou, ''HARD: Hardware-Assisted LockSet-Based Race Detection,'' *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 07), IEEE CS Press, 2007, pp. 121-132.
9. G.E. Suh et al., ''Secure Program Execution via Dynamic Information Flow Tracking,'' *Proc. Int'l Conf. Architectural Support for Languages and Operating Systems* (ASPLOS 04), ACM Press, 2004, pp. 85-96.
10. M. Dalton, H. Kannan, and C. Kozyrakis, ''Raksha: A Flexible Information Flow Architecture for Software Security,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 07), ACM Press, 2007, pp. 482-493.
11. G. Venkataramani et al., ''FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation,'' *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 08), IEEE CS Press, 2008, pp. 173-184.
12. S. Chen et al., ''Log-Based Architectures for General-Purpose Monitoring of Deployed Code,'' *Proc. First Workshop on Architectural and System Support for Improving Software Dependability,* ACM Press, 2006, pp. 63-65.
13. M.L. Corliss, E.C. Lewis, and A. Roth, ''DISE: A Programmable Macro Engine for Customizing Applications,'' *Proc. Int'l*

*Symp. Computer Architecture* (ISCA 03), 2003, IEEE CS Press, pp. 362-373.

14. J. Wilander and M. Kamkar, ''A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention,'' *Proc. Network and Distributed System Security Symp.* (NDSS 03), The Internet Society, 2003; http://www.isoc.org/isoc/conferences/ndss.

**Shimin Chen** is a research scientist at Intel Research Pittsburgh. His research interests include database systems, computer architecture, operating systems and distributed systems, with special emphasis on exploiting modern hardware features for software correctness, performance, and power efficiency. Chen has a PhD in computer science from Carnegie Mellon University.

**Michael Kozuch** is a principal researcher with Intel. His interests include processor microarchitecture and cluster architecture; he is particularly interested in the interaction of virtualization with these two topics. Kozuch has a PhD in electrical engineering from Princeton University.

**Theodoros Strigkos** is a PhD student in electrical and computer engineering at Carnegie Mellon University. His research interests include microarchitecture and multiprocessor architecture. Strigkos has an MS in electrical and computer engineering from Carnegie Mellon University. He is a student member of the ACM.

**Babak Falsafi** is a professor in the School of Computer and Communication Sciences and directs the Parallel Systems Architecture Laboratory at École Polytechnique Fédérale de Lausanne. His research targets architectural support for parallel programming, resilient systems, architectures to break the memory wall, and analytic and simulation tools for computer system performance evaluation. Falsafi has a PhD in computer science from University of Wisconsin at Madison. He is a senior member of the IEEE and the ACM.

**Phillip B. Gibbons** is a principal research scientist at Intel Research Pittsburgh. His research interests include parallel and distributed computing, databases, and sensor networks. Gibbons has a PhD in computer science from the University of California at Berkeley. He is a Fellow of the ACM.

**Todd C. Mowry** is a professor in the Computer Science Department at Carnegie Mellon University. He currently co-leads the Claytronics project and the Log-Based Architectures project. His research interests span a broad set of systems areas, as well as database performance and modular robotics. Mowry has a PhD in electrical engineering from Stanford University.

**Vijaya Ramachandran** is a professor of computer science at the University of Texas at Austin. Her research interests are in algorithm design and analysis, graph theory, data structures, and algorithms for multicore computing. Ramachandran has a PhD in electrical engineering and computer science from Princeton University. She serves on the Editorial Boards of *Journal of the ACM, SIAM Journal on Computing,* and *ACM Transactions on Algorithms.*

**Olatunji Ruwase** is a computer science PhD student at Carnegie Mellon University. His research interests are in compilers and computer architecture. Ruwase has an MSc in computer science from Stanford University.

**Michael Ryan** is a research engineer at Intel Research Pittsburgh. His interests include systems-level development, emulation, rapid application development, and cloud computing. Ryan has a BS in computer science from Carnegie Mellon University.

**Evangelos Vlachos** is a PhD student in electrical and computer engineering at Carnegie Mellon University. His research interests include computer architecture. Vlachos has an MSc in computer science from the University of Crete, Greece. He is a student member of the ACM.

Direct questions and comments about this article to Shimin Chen, Intel Research Pittsburgh, 4720 Forbes Ave., Ste. 410, Pittsburgh, PA 15213; shimin.chen@intel.com.