Behave or Be Watched: Debugging with Behavioral Watchpoints

Akshay Kumar
akshay@eecg.toronto.edu

Peter Goodman
pag@cs.toronto.edu

Ashvin Goel
asvhin@eecg.toronto.edu

Angela Demke Brown
demke@cs.toronto.edu

ABSTRACT

Finding, understanding, and fixing bugs in an operating system is challenging. Dynamic binary translation (DBT) systems provide a powerful facility for building program analysis and debugging tools. However, DBT abstractions are too low-level and provide limited contextual information for instrumentation tools.

We introduce *behavioral watchpoints*, a new software-based watchpoint framework that simplifies the implementation of DBT-based program analysis and debugging tools. Behavioral watchpoints extend the traditional approach of using a DBT system by providing context-specific information at the instruction level and specializing instrumentation to individual data structures. We describe four applications developed using our watchpoint framework: detecting buffer overflows, detecting read-before-write and memory freeing bugs, detecting memory leaks and enforcing fine-grained memory access policies. We implemented behavioral watchpoints using Granary, a new DBT framework. We show that the overheads are reasonable for their intended use in analyzing and debugging kernel modules.

1. INTRODUCTION

Program debugging is a tedious and time-consuming part of software development. Detecting software bugs such as data corruption, bad pointers, and data races requires developers to manually inspect millions of instructions that may modify data. The advent of multicore systems and the ever-increasing size and complexity of software has made debugging even more challenging.

Dynamic binary translation (DBT) systems provide a powerful facility for building debugging and analysis tools because they allow instrumenting the entire program at an instruction granularity [4]. For example, Valgrind's Memcheck [14] and Helgrind [11] use binary translation to detect common memory errors (e.g., use-after-free, read-before-write, memory leaks) and threading bugs (e.g., data races) in user space programs. These tools are powerful but developing them is challenging for the following three reasons.

First, a DBT system provides infrastructure for code-centric instrumentation, whereas many interesting debugging applications would prefer to have data-centric instrumentation. Applications such as data race detection, memory usage bugs, or performance debuggers that find false sharing hotspots are all naturally data-centric. Second, DBT abstractions are too low-level: individual instructions only reveal what memory addresses are being accessed. This makes it challenging to specialize instruction-level instrumentation to perform higher-level analysis. For example, developing a tool to debug the data corruption problem in a specific field of a data structure (e.g., `i_flags` field of a file `inode` structure), or detecting an invariant violation in a data structure requires instrumentation to be specialized to individual data structures. Third, existing DBT systems instrument all code to provide comprehensive coverage, which introduces significant overheads for realistic instrumentation. For example, if a memory corruption bug affects only `inode` structures, then instrumenting every memory access in the kernel is excessive. In practice, we would like to instrument only the code that operates on `inodes`.

In this paper, we introduce *behavioral watchpoints*, a novel software-based watchpoint framework that simplifies the implementation of DBT-based program analysis and debugging tools. Similar to previous software-based watchpoints [17], we support millions of watchpoints, enabling large-scale program analysis. However, unlike previous approaches that are limited by their view of memory as an opaque sequence of bytes, behavioral watchpoints embed *context-specific* information in each watchpoint, and this information is available when a watched address is accessed. Upon access, the watchpoint action taken depends on this context, implying that different watchpoints *behave* differently. For example, if a programmer wishes to debug corruption to a specific field of a data structure, such as a field in the TCP buffer header, then a behavioral watchpoint will be triggered only when the specific field is updated in any TCP buffer header. This approach simplifies building powerful DBT tools because the context-specific information can be arbitrarily rich (e.g., derived from static analysis), and is available as needed at runtime.

A key feature of behavioral watchpoints is that they enable adding instrumentation selectively, by enabling or disabling binary translation on demand, so that overhead is introduced only when instrumentation is needed. A watch-

HotDep '13, November 03-06, 2013, Farmington, PA, USA

ACM 978-1-4503-2457-1/13/11 ...$15.00.
http://dx.doi.org/10.1145/2524224.2524234

point can be triggered by a hardware trap that starts binary translation and watchpoint interpretation. The translation may continue until the end of the basic block or current function. This approach benefits from the lower overhead of binary translation when several watchpoints are likely to be triggered, and the lower overhead of infrequent traps when watchpoints are unlikely to be triggered. For example, kernel modules may initialize structures (such as the `sk_buff` structure used by network drivers) that are shared with the core kernel. The kernel expects such structures to contain legitimate data pointers when they are received from the module. However, a module can pass a bad pointer and cause the kernel to access illegal memory. Finding the source of this corruption requires complete visibility into all memory accesses to the structure, whether in the module or the kernel. Behavioral watchpoints enable this visibility with low overhead by comprehensive instrumentation of module code and on-demand translation of kernel code.

We have implemented behavioral watchpoints using Granary, a dynamic binary translation (DBT) framework designed to instrument kernel modules [5, 4]. Granary instruments arbitrary, binary Linux kernel modules efficiently and without imposing overhead when the core kernel is running. We have rapidly prototyped several module debugging tools using behavioral watchpoints. These tools include a buffer-overflow detector, a memory leak detector, and a shadow memory based tool for logging the access patterns of different types of modules for detecting buggy or malicious behavior. We describe these tools in more detail in Section 4.

2. DESIGN

A behavioral watchpoint is a software-based watchpoint that triggers the invocation of a function when any watched memory is accessed. Unlike most software-based watchpoints, a behavioral watchpoint watches a *range* of addresses, enabling object-granularity watchpoints (i.e., one watchpoint watches an entire object).

We implement behavioral watchpoints by adding an extra level of indirection to memory addresses. An unwatched address is converted into a watched address by changing its high-order bits. These high-order bits indirectly identify context-specific information about the range of memory being watched. This information, called the watchpoint's *descriptor*, contains the originating watched address, meta-information and a set of functions to invoke when watched memory is dereferenced. Since watchpoint information is embedded in the high-order bits, a typical offset of a watched address is another watched address that shares the same descriptor.

Our design is shown in Figure 1. It uses 15 high-order bits (called the *counter index*) and an additional 8 bits (bits 20-27, called the *inherited index*) of a watched address to identify the index into a global *watchpoint descriptor table* which stores the pointer to the watchpoint's descriptor. The key advantage of our watchpoint scheme is the ability to directly map watched addresses to unwatched addresses using a simple bitmask. One drawback of the scheme is that an offset of a watched address can cause the low-order bits to overflow into the inherited index. We overcome this issue by assigning multiple descriptors for the watched objects holding the same meta-information and by putting them in adjacent indices.

We take advantage of the x86-64 architecture for implementing watched addresses. In kernel space on x86-64, canonical addresses have their 16 high-order bits set to 1. Watched addresses do not take this form; they are non-canonical addresses that trigger a hardware exception when dereferenced. The watchpoint framework uses two approaches to perform memory operations on watched objects. First, when watchpoints are expected to be triggered frequently, it dynamically adds instrumentation at every memory load and store to avoid hardware exceptions. Watched addresses are detected before they are dereferenced and resolved to their unwatched counterparts (by masking the 16 high-order bits to 1). Second, when watchpoints are unlikely to be triggered, the alternative approach is to dereference a watched address and implement behavioral watchpoints in the trap handler. This enables on-demand binary translation and allows adding instrumentation only when a watchpoint gets triggered.

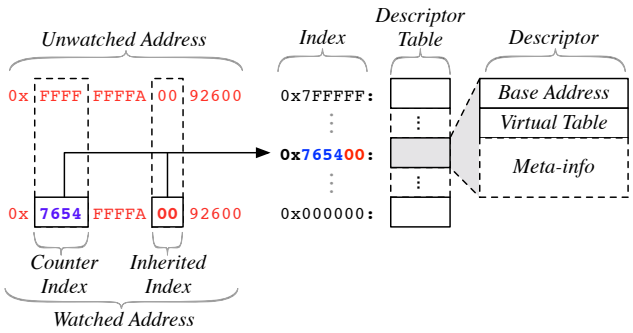

Figure 1: A watched address (bottom left) and its corresponding unwatched address (top left) are compared. The process of resolving the watchpoint descriptor is shown.

Our design separates the allocation and management of descriptors from the watchpoint framework. It is the responsibility of each client to manage its descriptors. When a watchpoint gets added to an object, the client determines the *vtable*, *type* and *meta-information* that needs to be stored in the descriptor, as shown in Figure 1. The vtable determines the function that is invoked when watched memory is accessed. Each vtable provides eight functions: four read and four write functions. Each function is specific to a memory operand size (1, 2, 4, or 8 bytes). A watchpoint descriptor is initialized with either a generic or a type-specific vtable, which is specific to the *type* of the watched address. The meta-information allows the descriptors to be arbitrarily customized or extended based on the needs of the client.

Our design allows the same range of memory to be watched differently. For example, two pointers to the same object can be watched separately, so long as they manage different descriptors. This feature is useful for distinguishing logically different objects that occupy the same memory. For example, this feature enables efficient detection of use-after-free bugs without preventing deallocated memory from being immediately reallocated for use. Having one watchpoint for the freed memory, and another watchpoint for newly allocated memory occupying the same space, is critical for this application.

Behavioral watchpoints can be used virally. If an address A is watched, then every address derived from A (e.g.,

through copying or offsetting) is also watched. This is useful for memory and taint analysis tools. For instance, a watchpoint that is added early in the lifetime of an address (e.g., immediately before the address of newly allocated memory is returned from an allocator) can persist and propagate until no more derived addresses exist.

3. IMPLEMENTATION

We implemented behavioral watchpoints using Granary, a dynamic binary translation (DBT) framework [5]. Granary instruments arbitrary, binary Linux kernel modules efficiently and without imposing overhead when the core kernel is running. Our aim is to use Granary to analyze and debug kernel modules, which are a frequent source of vulnerabilities in operating systems [3, 10].

Granary is unique among DBT systems because it analyzes and uses program type information. For example, Granary can substitute the execution of a function with a *wrapped* version of itself. A wrapped function has the same type specification as its unwrapped counterpart and can freely modify its arguments and return value. Granary can wrap some module functions in this way, even if the module source code is unavailable.

While Granary provides a framework for instrumenting kernel modules, we found it was hard to write powerful instrumentation code using low-level DBT abstractions, which motivated the design of behavioral watchpoints. Next, we describe examples of watchpoint-based debugging applications developed for kernel modules.

4. APPLICATIONS

The following sub-sections describe four applications of behavioral watchpoints and their implementations. In these sub-sections, we use the term *object* to refer to a range of memory locations that are allocated together.

4.1 Buffer Overflows

A buffer overflow occurs when a program—in an attempt to write to some object's memory—actually writes to adjacent memory cells. One method of detecting buffer overflows relies on the compiler to allocate "poisoned" regions of memory around each object [13]. Small overflows (e.g., off-by-one errors) are detected by this approach because they access poisoned memory. Big overflows that "skip" over poisoned memory and access nearby objects in memory are not detected.

Our insight is that unrelated objects will have different base addresses (i.e., the address of one object will not be derived from the base address of another), and thus each object can be distinguished and uniquely identified with a separate watchpoint address

We employ two overflow detection policies: heap-based, and stack-based.

Heap-based overflow detection.

The heap-based detection policy detects buffer overflow errors on all heap-allocated objects. We use Granary to wrap the kernel's memory allocators (e.g., `kmalloc`) and add watchpoints to the addresses returned by those allocators, as shown in Figure 2. The lifetime of an added watchpoint is tied to the lifetime of the memory it watches. Each watchpoint's descriptor records bounds information about the allocated memory in the form of the object's base and limit address [8]. A buffer overflow is detected when a dereference of a watched address occurs outside of the bounds recorded by the watchpoint's descriptor. The memory operand-size-specific vtable functions help catch corner cases where memory reads or writes access both an object and its adjacent memory cells.

```
FUNC_WRAPPER(__kmalloc, (size, flags), {
  void *addr = __kmalloc(size, flags);
  ADD_WATCHPOINT(addr, size);
  return addr;
})
```

Figure 2: Definition of the `__kmalloc` function wrapper in Granary. The above code expands into a function for wrapping the `__kmalloc` allocator. Calls to `__kmalloc` are transparently substituted with calls to the generated wrapper. The wrapper invokes the original `__kmalloc` function and returns a watched version of the allocated address.

Stack-based overflow detection.

To detect stack overflows, we view the memory occupied by the activation frame of an invoked function as a dynamically-sized buffer. We associate a descriptor with the buffer represented by the activation frame of a called function. This descriptor tracks the bounds of the frame over the lifetime of the function call.

We update the bounds of the frame (in the descriptor) when the frame grows or shrinks. When a function returns, the descriptor's bounds shrink to zero, but the descriptor remains allocated. We detect the two most common sources of stack overflows. First, if we see an instruction that copies the stack or frame pointers, then we assume that the copied address can escape the function. A stack address escaping a function is a potential stack-overflow risk. Adding the frame's watchpoint to this address *taints* the copied address. Future copies or displacements of the watched address implicitly propagate its taintedness because offsets of a watched address reference the same descriptor. A dereference of an escaped pointer—even one happening after the function has returned—is detected as an overflow because the watchpoint descriptor remains live. Second, if we see an indexed dereference of the stack or frame pointers that uses a dynamically bound index, then we assume that the effective memory address accessed is a potential stack-overflow risk. We instrument the dereferencing instruction to add the frame's watchpoint to the effective address before the address is dereferenced.

4.2 Selective Memory Shadowing

In this section, we show how to use behavioral watchpoints to shadow memory. Previous work has focused on full memory shadowing [12], while watchpoints enable *selective* shadowing of watched objects. We describe how the initialization state of each byte of watched memory is tracked using shadow memory, and how to detect bugs related to the usage of heap-allocated memory.

We use Granary to wrap kernel memory allocators and deallocators (e.g., `kmalloc`, `kfree`). Wrapped allocators add watchpoints to allocated memory, and wrapped deallocators remove watchpoints before invoking the kernel's

deallocators. Selective shadow memory is maintained as a watchpoint descriptor-specific, variable-sized bitset. Each byte of allocated memory corresponds to one bit of shadow memory. The bits in shadow memory are initialized to zero. Individual bits are flipped to one by injected instrumentation code when a write occurs to the memory shadowed by those bits.

Read-before-write bugs.

Memory read instrumentation detects read-before-write bugs by checking if the shadow bit corresponding to one of the read bytes is zero. However, this method of detection can report false positives: it is common for larger-than-needed reads to be performed and for compiler-added structure padding to be read (but never written). A relaxed read-before-write memory checking policy requires that at least one shadow bit is set for every read operation.

Memory freeing bugs.

An invalid free bug occurs when an object passed to a deallocator was not previously obtained from an allocator. For example, this may happen if an offset of an allocated object is freed instead of the original object. We detect invalid frees when the freed address doesn't match the base address of the watched object. We detect use-after-free bugs by marking the descriptor of a watched address being deallocated as dead. The lifetime of a dead descriptor extends beyond that of the object so that a later use of any watched address with that descriptor will report a use-after-free bug. We detect double-free bugs when the descriptor of a watched address being deallocated is already marked as dead.

4.3 Memory Leak Detection

We built a memory leak detector for module-owned objects. A module is responsible for deallocating such objects. Not all module-owned objects are allocated by modules. For example, `sk_buff` objects used by network modules are allocated by kernel interrupt handlers, but must be deallocated by the network modules. Our approach ensures that all module-owned objects are watched when allocated.

The leak detector scans kernel memory using the conventional mark/sweep algorithm [2] to detect if/when module-owned objects become unreachable starting from a set of "root" objects. This is challenging because modules regularly lose internal references to the objects that they own. For example, a network module can allocate objects and pass them t othe kernel with the `net_device` structure, without retaining references to those objects. The network module can later indirectly access these objects using the kernel's `netdev_priv` interface. Tracking the liveness of these objects requires a view on kernel execution not normally provided by module-only instrumentation. Behavioral watchpoints solve this problem by giving the leak detector visibility into kernel accesses to watched objects, which trigger hardware traps that attach instrumentation to the kernel code. This instrumentation marks kernel-accessed watched objects as live.

The leak detector benefits from using watchpoints in three ways: i) watched addresses are easily disambiguated from normal memory addresses and integers that look like watched addresses; ii) descriptors provide meta-information that allows our system to limit the scope of scanning; and iii) scanning can stop if all watched objects are reached.

```
// Invariant: rtc_fops->ioctl == &rtc_ioctl
WATCH_WRITE(struct file_operations, ioctl, {
  if(&rtc_fops  == base_address
  && &rtc_ioctl != ioctl) {
    // potential attack: prevent an anti-
    // virus scan from being scheduled!
  }
})
```

Figure 3: This code example shows how to check invariant 1(h) from [1] using field-accessor API. The invariant checked prevents a (potential) kernel rootkit from installing its own `ioctl` handler into the Real-Time Clock and preventing virus-scans from being scheduled, thus allowing rootkit to go undetected.

4.4 Field-grained Access Policies

We briefly describe the use of watchpoints at the granularity of object fields. Using a field-accessor API, we can detect Linux kernel rootkits, which is challenging because rootkits actively try to hide their existence. However, rootkits often leave hints of their presence in the form of violated data structure invariants [1, 7]. Figure 3 shows an example use of the field-accessor API that checks an invariant on the `ioctl` field of any watched object whose type is `struct file_operations`. A rootkit violating this invariant can prevent an anti-virus program from scheduling periodic scans—scans which might otherwise detect the presence of a rootkit. However, a violation of this invariant is immediately detected by our run-time system because the field-accessor API enables active rather than passive checking of invariants.

5. EVALUATION

We evaluated behavioral watchpoints using a microbenchmark and measured the performance of common filesystem operations on an in-memory disk. Our tests ran on a desktop equipped with an Intel® Core™ 2 Duo 2.93 GHz CPU with 4GB physical memory. The microbenchmark performed a tight-loop of memory operations and exhibits the worst-case CPU overhead of the baseline watchpoints instrumentation (3.8×). For the same microbenchmark, the buffer-overflow detector's overhead was ≈21×, which is caused by the instrumentation needed for precise bounds checking.

IOzone.

The IOzone benchmark creates two processes (one reader and one writer) that perform common file operations on a file of size 480Mb with a record size of 1Kb. In our experimental setup, we mounted the `ext3` filesystem on a 1GB RAMDisk to eliminate the high overhead of disk I/O, which would hide the CPU overhead of adding instrumentation. We also ensured that all operations go through the file system module code by enabling direct I/O to avoid the effect of the buffer cache. The mount operation loads two kernel modules: the `ext3` filesystem module and the `jbd` journaling module. Watchpoints were added to addresses returned by the two most-used allocators in `ext3` and `jbd`: `__kmalloc` and `kmem_cache_alloc`.

Figure 4 shows the drop in throughput of filesystem operations relative to native execution when using Granary and the watchpoints instrumentation. The *Granary* bar shows

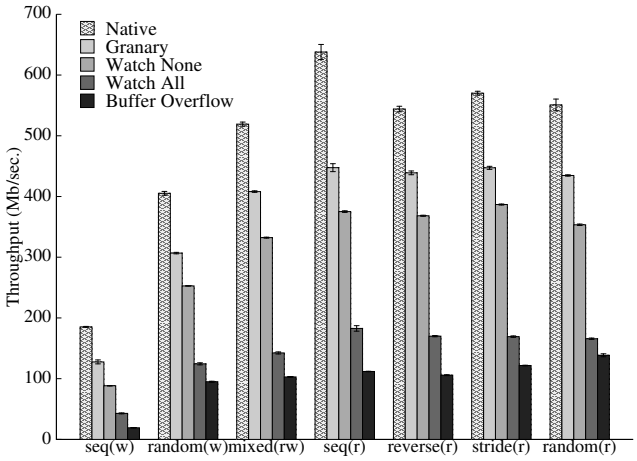

Figure 4: Throughput (in Mb/sec) for common file system operations (write workloads (w), read workloads (r)).

the cost of the dynamic binary instrumentation framework on which we build watchpoints. The *Watch None* bar represents the overhead of baseline watchpoints instrumentation with no added watchpoints. In this configuration, module code has added instrumentation to check each memory access instruction for watched addresses, however, no additional action is needed since no objects are watched. The *Watch All* bar shows the overhead where all module-allocated objects are watched. In this case, we see two costs. First, instrumented module code must check for watched addresses and convert them to their unwatched counterparts to perform the memory access. Second, when uninstrumented kernel code accesses a watched address, we must handle the resulting fault and perform some limited instrumentation around the faulting instruction. Finally, Figure 4 also shows the overhead of the heap-allocated buffer overflow detector using behavioral watchpoints. This case includes the cost of executing the appropriate watchpoint vtable function to check the bounds on each watched memory access.

The overhead of watching all module-allocated objects is high (≈70%) because many watched objects are accessed by uninstrumented kernel code, which results in many hardware exceptions. Our system recovers from these exceptions by instrumenting kernel code on-demand. We discovered that `inodes` are frequently accessed by the kernel (thus causing many exceptions) and that not watching `inodes` improves the performance of *Watch All* to be near that of *Watch None*. We also observed that the added cost of *Buffer Overflow* is smaller than expected, considering the additional work that it performs. This result again suggests that the overhead in *Watch All* is dominated by the cost of faults and our recovery mechanism, thereby masking the cost of the bounds checking. We are exploring strategies to automatically adapt the granularity of instrumentation to reduce repeated faults in core kernel code.

6. RELATED WORK

Greathouse *et al.* [6] propose a hardware solution that efficiently supports an unlimited number of watchpoints. Witchel and Asanovic [16] describe the implementation of memory protection domains for the Linux kernel. Protection domains are implemented using specialized hardware and enable fine- and coarse-grained memory protection using a mechanism similar to hardware watchpoints. Unlike our approach, both of these depend on specialized hardware and require that applications using this hardware separately maintain context-specific information. Suh *et al.* [15] propose a method of secure program execution by tracking dynamic information flow. Memory tagging at the hardware level allows their system to track tainted data as it propagates through a running program. Behavioral watchpoints are similar insofar as a watched address is tagged, and this tag propagates through a program.

Zhao *et al.* [17] describe a method of implementing an efficient and scalable DBT-based watchpoint system. It uses page protection and indirection through a hash table to track watched memory. This approach supports neither watching ranges of memory, nor context-specific information. Lueck *et al.* [9] introduce semantic watchpoints as part of the PinADX system. It enables interactive debugging by triggering debugger breakpoints when semantic conditions are met. While similar in spirit to behavioral watchpoints, semantic watchpoints do not maintain context-specific, per-watchpoint state.

7. CONCLUSIONS AND FUTURE WORK

We created behavioral watchpoints to simplify the implementation of DBT-based program analysis and debugging tools. Behavioral watchpoints address both the scalability limits of hardware watchpoints and the lack of contextual information with traditional software watchpoints. The key innovation is to maintain context-specific information about the memory being watched, which is required for large-scale program analysis, with the watchpoint itself. Thus, behavioral watchpoints resolve the incongruency between how existing software implements watchpoints and how program analysis tools use watchpoints. We demonstrated the usability of behavioral watchpoints by describing several memory error detection and protection applications. Although our current evaluation is limited, the overheads of behavioral watchpoints are quite reasonable for the target use of debugging kernel modules.

As future work, we will investigate the applicability of behavioral watchpoints for isolating and enforcing control-flow integrity policies on Linux kernel modules.

8. REFERENCES

[1] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Trans. Dependable Secur. Comput.*, 2011.

[2] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 157–164, New York, NY, USA, 1991. ACM.

[3] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-Granularity Software Fault Isolation. In *SOSP*, 2009.

[4] P. Feiner, A. D. Brown, and A. Goel. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *ASPLOS*, 2012.

[5] P. Goodman, A. Kumar, A. D. Brown, and A. Goel. Granary: Comprehensive kernel module instrumentation. Poster at OSDI'12, 2012.

[6] J. L. Greathouse, H. Xin, Y. Luo, and T. M. Austin. A case for unlimited watchpoints. In *ASPLOS*, 2012.

[7] O. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *ASPLOS*, 2011.

[8] S. Kendall. Bcc: Run-time checking for C programs. In *USENIX Toronto 1983 Summer Conference Proceedings*, 1983.

[9] G. Lueck, H. Patil, and C. Pereira. PinADX: an interface for customizable debugging with dynamic instrumentation. In *CGO*, 2012.

[10] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *SOSP*, 2011.

[11] A. Müehlenfeld and F. Wotawa. Fault detection in multi-threaded c++ server applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 142–143, New York, NY, USA, 2007. ACM.

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

[13] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*, 2012.

[14] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

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

[16] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *SOSP*, pages 31–44, 2005.

[17] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: efficient debugging using dynamic instrumentation. In *CC / ETAPS*, 2008.